TABLE OF CONTENTS

PREFACE

This manual is about the ATARI Personal Computer System. It covers both the ATARI 400™ and the ATARI 800™ computers. These two computers are electrically identical, differing only in mechanical features such as the keyboards and cartridge slots. The purpose of this manual is to explain in detail how to use all the features of the ATARI Personal Computer System. This is a complex and powerful machine, and the explanations are accordingly rather long. Furthermore, they demand some expertise on the part of the reader. This book is not intended for the beginning programmer. The reader should be thoroughly familiar with the BASIC Reference Manual provided with the computer. Familiarity with assembly language is also essential. A glossary in the back defines and explains some of the less commonly encountered jargon. However, this glossary does not include terms that every serious personal computer programmer should already know.

This book was written as a training manual for professional programmers wishing to use the ATARI Personal Computer System. At some later date it may be modified for general use. It does not supplant the technical reference manual (ATARI part number C016555). That manual is a reference manual; it is very useful for programmers who already understand the system. This book is intended to be a tutorial; it will explain ideas and possibilities rather than define registers and control codes.

The Software Development Support Group wrote this book. Chris Crawford wrote chapters 1 through 6 and Appendices I and II. Lane Winner wrote chapter 7 and Appendix IV with assistance from Jim Cox. Amy Chen wrote Appendix III. Mike Ekberg wrote chapters 8 and 9 with assistance from John Eckstrom. Kathleen Pitta wrote Appendix X. Bob Fraser wrote Chapter 10. Gus Makreas prepared the glossary and table of contents. The final result has many flaws, but we are proud of it.

John Eckstrom

Michael A. Ekberg

Gus Makreas

Lane Winner

Elizabeth Herndon

Kathleen Pitta

Jim Cox

Bob Fraser

Chris Crawford

CHAPTER 1
SYSTEM OVERVIEW

The ATARI Personal Computer System is a second generation personal computer. First and foremost, it is a consumer computer. The thrust of the design is to make the consumer comfortable with the computer. This consumer orientation manifests itself in many ways. First, the machine is idiot-proofed; the consumer is protected from mistakes by such things as polarized connectors that will not go in the wrong way, a power interlock on the lid to the internal electronics, and a pair of plastic shields protecting the SYSTEM RESET key. Second, the machine has a great deal of graphics power; people respond to pictures much more readily than to text. Third, the machine has strong sound capabilities; again, people respond to direct sensory input better than to indirect textual messages. Finally, the computer has joysticks and paddles for more direct tactile input than is possible with keyboards. The point here is not that the computer has lots of features but rather that the features are all part of a consistent design philosophy aimed directly at the consumer. The designer who does not appreciate this fundamental fact will find himself working against the grain of the system.

The internal layout of the ATARI 400/800™ computer is very different from other systems. It of course has a microprocessor (a 6502), RAM, ROM, and a PIA. However, it also has three special purpose LSI chips known as ANTIC, POKEY, and CTIA. These chips were designed by Atari engineers primarily to take much of the burden of housekeeping off of the 6502, thereby freeing the 6502 to concentrate on computations. While they were at it, they designed a great deal of power into these chips. Each of these chips is almost as big (in terms of silicon area) as a 6502, so the three of them together provide a tremendous amount of power. Mastering the ATARI 400/800 is primarily a matter of mastering these three chips.

ANTIC is a microprocessor dedicated to the television display. It is a true microprocessor; it has an instruction set, a program (called the display list), and data. The display list and the display data are written into RAM by the 6502. ANTIC retrieves this information from RAM using DMA. It processes the higher level instructions in the display list and translates these instructions into a real-time stream of simple instructions to CTIA.

CTIA is a television interface chip. ANTIC directly controls most of CTIA's operations, but the 6502 can be programmed to intercede and control some or all of CTIA's functions. CTIA converts the digital commands from ANTIC (or the 6502) into the signal that goes to the television. CTIA also adds some factors of its own, such as color values, player-missile graphics, and collision detection.

POKEY is a digital I/O chip. It handles such disparate tasks as the serial I/O bus, audio generation, keyboard scan, and random number generation. It also digitizes the resistive paddle inputs and controls maskable interrupt (IRQ) requests from peripherals.

All four of these LSI chips function simultaneously. Careful separation of their functions in the design phase has minimized conflicts between the chips. The only hardware level conflict between any two chips in the system occurs when ANTIC needs to use the address and data busses to fetch its display information. To do this, it halts the 6502 and takes control of the busses.

As with all 6502 systems, the I/O is memory-mapped. Figure 1.1 presents the coarse memory map for the computer. Figure 1.2 shows the hardware arrangement.

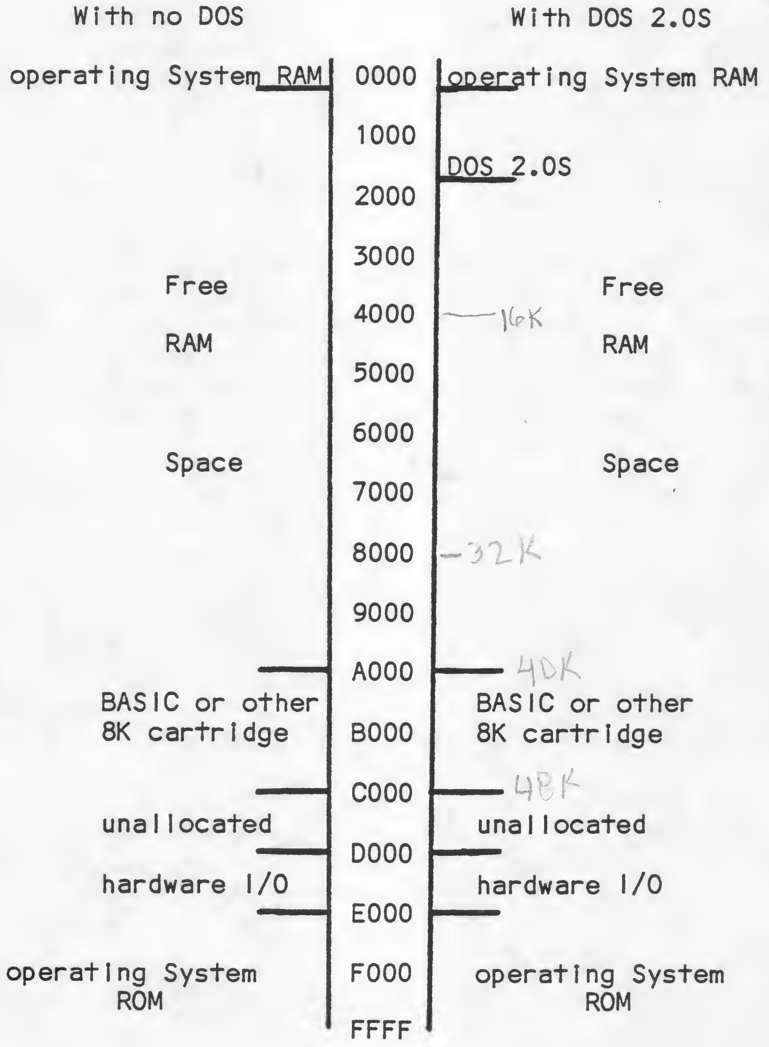
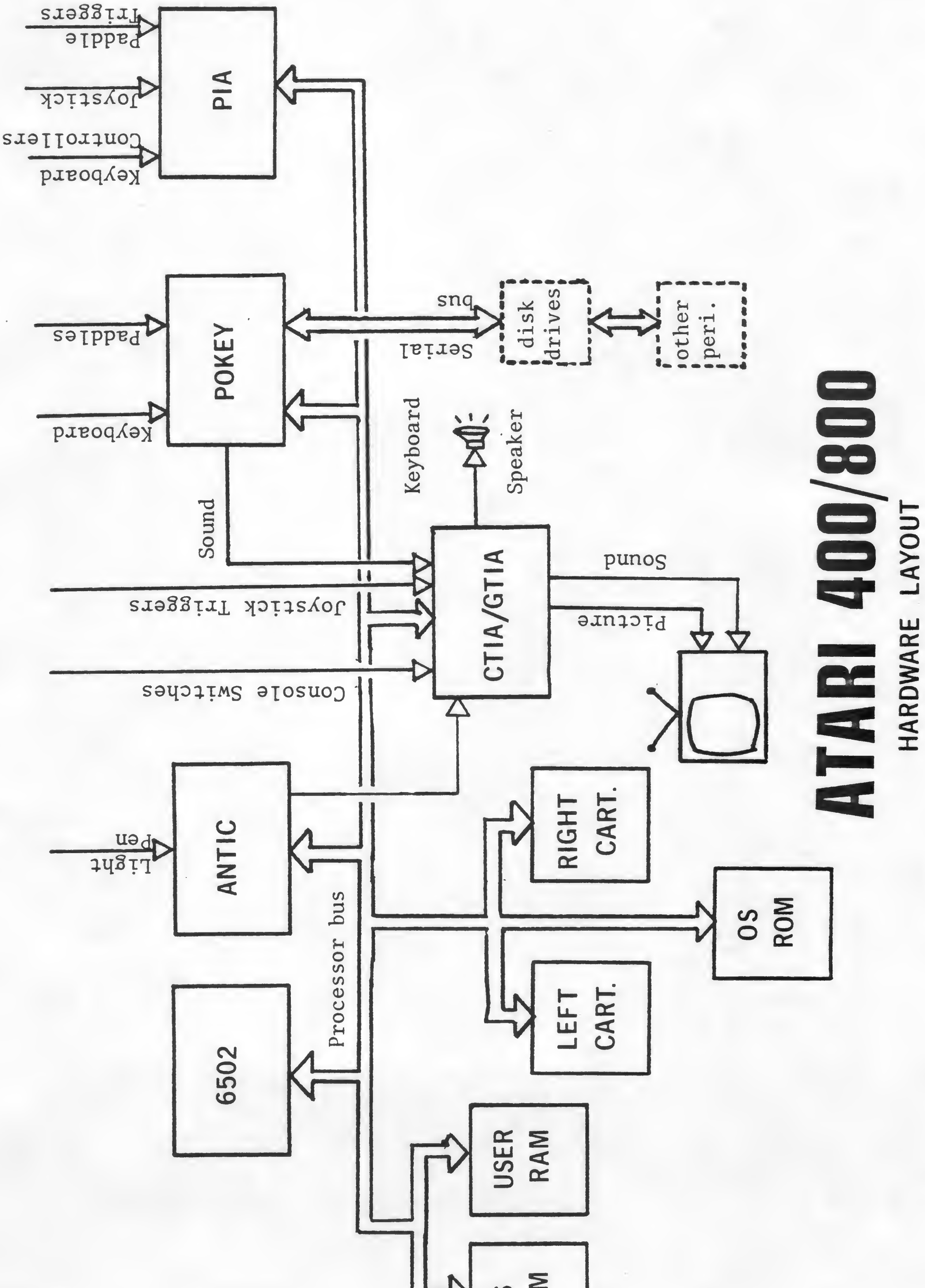


Figure 1.1
Coarse Memory Map for ATARI 400/800



ATARI 400/800
HARDWARE LAYOUT

CHAPTER 2
ANTIC AND THE DISPLAY LIST

TELEVISION DISPLAYS

In order to understand the graphics capabilities of the ATARI Personal Computer System one must first understand the rudiments of how a television set works. Television sets use what is called a raster scan display system. An electron beam is generated at the rear of the television tube and shot toward the television screen. Along the way, it passes between sets of horizontal and vertical coils which, if energized, can deflect the beam. In this way the beam can be made to strike any point on the screen. The electronics inside the television set cause the beam to sweep across the television screen in a regular fashion. The beam's intensity can also be controlled. If you make the beam more intense the spot in the screen that it strikes will glow brightly; if you make it less intense the spot will glow dimly or not at all.

The beam starts at the top left corner of the screen and traces horizontally across the screen. As it sweeps across the screen, its intensity changes will paint an image on the screen. When it reaches the right edge of the screen, it is turned off and brought back to left side of the screen. At the same time it is moved down just a notch. It then turns back on and sweeps across the screen again. This process is repeated for a total of 262 sweeps across the screen. (There are actually 525 sweeps across the screen in an alternating system known as interlace. We will ignore interlace and act as if the television has only 262 lines.) These 262 lines fill the screen from top to bottom. At the bottom of the screen (after the 262nd line is drawn), the electron beam is turned off and returned to the upper left corner of the screen. Then it starts the cycle all over again. This entire cycle happens 60 times every second.

Now for some jargon: a single trace of the beam across the screen is called a 'horizontal scan line'. A horizontal scan line is the fundamental unit of meausurement of vertical distance on the screen. You state the height of an image by specifying the number of horizontal scan lines it spans. The period during which the beam returns from the right edge to the left edge is called the 'horizontal blank'. The period during which the beam returns to the top of the screen is called the 'vertical blank'. The entire process of drawing a screen takes 16,684 microseconds. The vertical blank period is about 1400 microseconds. The horizontal blank takes 14 microseconds. A single horizontal line takes 64 microseconds.

Most television sets are designed with 'overscan'; that means that they spread the image out so that the edges of picture are off the edge of the television tube. This guarantees that you have no unsightly borders in your TV picture. It is very bad for computers, though, because screen information that is off the edge of the picture does you no good. For this reason the picture that the computer puts out must be somewhat smaller than what the television can theoretically display. For this reason only 192 horizontal scan lines are normally used by the Atari display. Thus, the normal limit of resolution of a television set used with this computer is 192 pixels vertically. The standard unit of horizontal distance is the 'color clock'. You specify the width of an image by stating how many color clocks wide it is. There are 228 color clocks in a single horizontal scan line, of which a maximum of 176 are actually visible. Thus, the ultimate limit for full color horizontal resolution with a standard color television is 176 pixels. It is possible with the ATARI Personal Computer System to go even finer and control individual half-clocks. This gives a horizontal resolution of 352 pixels. However, use of this feature will produce interesting color effects known as color artifacts. Color artifacts can be a nuisance if they are not desired; they can be a boon to the programmer who desires additional color and is not fazed by their restrictions.

COMPUTERS AND TELEVISIONS

The fundamental problem any microcomputer has in using a raster scan television for display purposes is that the television display is a dynamic process; because of this the television does not remember the image. Consequently, the computer must remember the screen image and constantly send a signal to the television telling it what to display. This process of sending information to the television is a continuous process and it requires full-time attention. For this reason most microcomputers have special hardware circuits that handle the television. The basic arrangement is the same on virtually all systems:

```
microprocessor--->screen RAM--->video hardware--->TV screen
```

The microprocessor writes information to the screen RAM area that holds the screen data. The video hardware is constantly dipping into this RAM area, getting screen data which it converts into television signals. These signals go to the television which then displays the information. The screen memory is mapped onto the screen in the same order that it follows in RAM. That is, the first byte in the screen memory maps to the top left corner of the screen, the second byte maps one position to the right, then the third, the fourth, and so on to the last byte which is mapped to the lower right corner of the screen.

The quality of the image that gets to the screen depends on two factors: the quality of the video hardware, and the quantity of screen RAM used for the display. The simplest arrangement is that used by TRS-80 and PET. (TRS-80 is a trademark of Radio Shack Co; PET is a trademark of Commodore Business Machines.) Both of these machines allocate a specific 1K of RAM as screen memory. The video hardware circuits simply pull data out of this area, interpret it as characters (using a character set in ROM), and put the resulting characters onto the screen. Each byte represents one character, allowing a choice of 256 different characters in the character set. With 1K of screen RAM one thousand characters can be displayed on the screen. There isn't much that can be done with this arrangement. The Apple uses more advanced video hardware. (Apple is a trademark of Apple Computers.) Three graphics modes are provided: text, lo-res graphics, and hi-res graphics. The text graphics mode operates in much the same way that the PET and TRS-80 displays operate. In the low-resolution graphics mode, the video hardware reaches into screen memory and interprets it differently. Instead of interpreting each byte as a character, each byte is interpreted as a pair of color nybbles. The value of each nybble specifies the color of a single pixel. In the high-resolution graphics mode each bit in screen memory is mapped to a single pixel. If the bit is on, the pixel gets color in it; if the bit is off, the pixel stays dark. The situation is complicated by a variety of design nuances in the Apple, but that is the basic idea. The important idea is that the Apple has three display modes; three completely different ways of interpreting the data in screen memory. The Apple video hardware is smart enough to interpret a screen memory byte as either an 8-bit character (text mode), two 4-bit color nybbles (lo-res mode), or 7 individual bits for a bit map (hi-res mode).

ANTIC, A VIDEO MICROPROCESSOR

The ATARI 400/800™ display list system represents a generalization of these systems. Where PET and TRS-80 have one mode and Apple has three modes, the ATARI 400/800™ has 14 modes. The second important difference is that display modes can be mixed on the screen. That is, the user is not restricted to a choice between a screenful of text or a screenful of graphics. Any collection of the 14 graphics modes can be displayed on the screen. The third important difference is that the screen RAM can be located anywhere in the address space of the computer and moved around while the program is running, while the other machines use fixed screen RAM areas.

All of this generality is made possible by a video microprocessor called ANTIC. Where the earlier systems used rather simple video circuitry, Atari designed a full-scale microprocessor just to handle the intracacies of the television display. ANTIC is a true microprocessor; it has an instruction set, a program, and data. The program for ANTIC is called the display list. The display list specifies three things: where the screen data may be found, what display modes to use to interpret the screen data, and what special display options (if any) should be implemented. When using the display list, it is important to shed the old view of a screen as a homogeneous image in a single mode and see it instead as a stack of 'mode lines'. A mode line is a collection of horizontal scan lines. It streches horizontally all the way across the screen. A Graphics 2 mode line is 16 horizontal scan lines high, while a Graphics 7 mode line is only two scan lines high. Many Graphics modes available from BASIC are homogeneous; an entire screen of a single mode is set up. One must not limit her imagination to this pattern; with the display list you can create any sequence of mode lines down the screen. The display list is a collection of code bytes which specify that sequence.

ANTIC'S instruction set is rather simple. There are four classes of instructions: map mode instructions, character mode instructions, blank line instructions, and jump instructions. Map mode instructions cause ANTIC to display a mode line with simple colored pixels (no characters). Character mode instructions cause ANTIC to display a mode line with characters in it. Blank line instructions cause ANTIC to display a number of horizontal scan lines with solid background color. Jump instructions are analogous to a 6502 JMP instruction; they reload ANTIC's program counter. There are also four special options that can sometimes be specified by setting a designated bit in the ANTIC instruction. These options are: display list interrupt (DLI), load memory scan (LMS), vertical scroll, and horizontal scroll.

Map mode instructions cause ANTIC to display a mode line containing pixels with solid color in them. The color displayed comes from a color register. The choice of color register is specified by the value of the screen data. In four-color map modes (BASIC modes 3, 5, and 7, and ANTIC modes 8, A, D, and E), a pair of bits is required to specify a color:

value of bit pair		color register used
00	0	COLBAK
01	1	COLPF0
10	2	COLPF1
11	3	COLPF2

Since only two bits are needed to specify one pixel, 4 pixels are encoded in each screen data byte. For example, a byte of screen data containing the value $1B would display four pixels; the first would be the background, the second would be color register 0, the third would be color register 1, and the fourth would be color register 2:

$1B = 00011011 = 00 01 10 11

In two-color map modes (BASIC modes 4, 6, and 8, and ANTIC modes 9, B, C, and F) each bit specifies one of two color registers. A bit value of 0 selects background color for the pixel and a bit value of 1 selects color register 0 for the pixel. Eight pixels can be stored in one screen data byte.

There are eight different map display modes. They differ in the number of colors they display (2 vs 4), the vertical size one mode line occupies (1 scan line, 2, 4, or 8), and the number of pixels that fit horizontally into one mode line (40, 80, 160, or 320). Thus, some map modes give better resolution; these will of course require more screen RAM. Figure 2.1 presents this information for all modes:

ANTIC mode	BASIC mode	number colors	scan lines/ mode line	pixels/ mode line	bytes/ line	bytes/ screen
2	0	2	8	40	40	960
3	none	2	10	40	40	760
4	none	4	8	40	40	960
5	none	4	16	40	40	480
6	1	5	8	20	20	480
7	2	5	16	20	20	240
8	3	4	8	40	10	240
9	4	2	4	80	10	480
A	5	4	4	80	20	960
B	6	2	2	160	20	1920
C	none	2	1	160	20	3840
D	7	4	2	160	40	3840
E	none	4	1	160	40	7680
F	8	2	1	320	40	7680

Figure 2.1
ANTIC mode line requirements

Character mode instructions cause ANTIC to display a mode line with characters in it. Each byte in screen RAM specifies one character. There are six character display modes. Character displays are discussed in Chapter 3.

Blank line instructions produce blank lines with solid background color. There are eight blank line instructions; they specify skipping one through eight blank lines.

There are two jump instructions. The first (JMP) is a direct jump; it reloads ANTIC's program counter with a new address which follows the JMP instruction as an operand. Its only function is to provide a solution to a tricky problem: ANTIC's program counter has only ten bits of counter and six bits of latch and so the display lit cannot cross a 1K boundary. If the display list must cross a 1K boundary then it must use a JMP instruction to hop over the boundary. Note that this means that display lists are not fully relocatable.

The second jump instruction (JVB) is more commonly used. It reloads the program counter with the value in the operand and waits for the television to perform a vertical blank. This instruction is normally used to terminate a display list by jumping back up to the top of the display list. Jumping up to the top of the display list turns it into an infinite loop; waiting for vertical blank insures that the infinite loop is synchronized to the display cycle of the television. Both JMP and JVB are three byte instructions; the first byte is the opcode, the second and third bytes are the address to jump to (lo then hi).

The four special options mentioned previously will be discussed in Chapters 5 and 6. The load memory scan (LMS) option must have a preliminary explanation. This option is selected by setting bit 6 of a map mode or a character mode instruction byte. When ANTIC encounters such an instruction, it will load its memory scan counter with the following two bytes. This memory scan counter tells ANTIC where the screen RAM is. It will begin fetching display data from this area. The LMS instruction is a three byte instruction: one byte opcode followed by two bytes of operand. In simple display lists the LMS instruction is used only once, at the beginning of the display list. It may sometimes be necessary to use a second LMS instruction. The need arises when the screen RAM area crosses a 4K boundary. The memory scan counter has only 12 bits of counter and 4 bits of latch; thus, the display data cannot cross a 4K boundary. In this case an LMS instruction must be used to jump the memory scan counter over the boundary. Note that this means that display data is not fully relocatable. LMS instructions have wider uses which will be discussed later.

BUILDING DISPLAY LISTS

Every display list should start off with 3 'blank 8 lines' instructions. This is to defeat vertical overscan by bringing the beginning of the display 24 scan lines down. After this is done the first display line should be specified. Simultaneously, the LMS should be used to tell ANTIC where it will find the screen RAM. Then follows the display list proper, which lists the display bytes for the mode lines on the screen. The total number of horizontal scan lines produced by the display list should always be 192 or less; ANTIC does not maintain the screen timing requirements of the television. If you give ANTIC too many scan lines to display it will do so, but the television screen will probably roll. Displaying fewer than 192 scan lines will cause no problems; indeed, it will decrease 6502 execution time by reducing the number of cycles stolen by ANTIC. The programmer must calculate the sum of the horizontal scan lines produced by her display list and verify it herself. The display list terminates with a JVB instruction. Here is a typical display list for a standard BASIC Graphics mode 0 display (all values are in hexadecimal):

```
70   blank 8 lines
70   blank 8 lines
70   blank 8 lines
42   display ANTIC mode 2 (BASIC mode 0)
20   also, screen memory starts at 7C20
7C
02   display ANTIC mode 2
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
41   jump and wait for vertical blank
E0   to display list which starts at
7B   $7BE0
```

As you can see, this display list is short---only 32 bytes. Most display lists are less than 100 bytes long. Furthermore, they are quite simple in structure and easy to set up.

To implement your own display list you must first design the display format. This is best done on paper. Lay out the screen image and translate it into a sequence of mode lines. Keep track of the scan line count of your display by looking up the scan line requirements of the various modes in Figure 2.1. Translate the sequence of mode lines into a sequence of ANTIC mode bytes. Put three 'blank 8 lines' bytes ($70) at the top of the list. Set bit 6 of the first display byte (that is, make the upper nybble a 4). This makes a load memory scan command. Follow with two bytes which specify the address of the screen RAM (lo then hi). Then follow with the rest of the display bytes. At the end of your display list put in the JVB instruction ($41) and the address of the top of the display list. Now store all of these bytes into RAM. They can be anywhere you want; just make sure they don't overlay something else and your JVB points to the top of the display list. The display list must not cross a 1K address boundary. If you absolutely must have it cross such a boundary, insert a JMP instruction just in front of the boundary. The JMP instruction's operand is the address of the first byte on the other side of the boundary. Next you must turn off ANTIC for a fraction of a second while you rewrite its display list pointer. Do this by writing a 0 into SDMCTL at location $22F. Then store the address of the new display list into $230 and $231 (lo then hi). Lastly, turn ANTIC back on with a $22 into SDMCTL. During the vertical blank, while ANTIC is quiet, the operating system will reload ANTIC's program counter with these values.

WRITING TO A CUSTOM DISPLAY LIST SCREEN

Screen memory can be placed anywhere in the address space of the computer. Normally the display list specifies the beginning of the screen memory with the first display instruction---the initial LMS instruction. However, ANTIC can execute a new LMS instruction with each display line of the display list if this is desired. In this way information from all over the address space of the computer can be displayed on a single screen. This can be of value in setting up independent text windows.

There are several restrictions in your placement of the screen memory. First, screen memory cannot cross a 4K address boundary. If you cannot avoid crossing a 4K boundary (as would be the case in BASIC mode 8, which uses 8K of RAM) you must reload the memory scan counter with a new LMS instruction. Second, if you wish to use any of the operating system screen routines you must abide by the conventions the OS uses. This can be particularly difficult when using a modified display list in a BASIC program. If you alter a standard display list from a BASIC program and then attempt to PRINT or PLOT to the screen, the OS will do so under the assumption that the display list is unchanged. This will probably result in a garbled display.

There are three ways the display can fail when you attempt this. First, BASIC may refuse to carry out a screen operation because it is impossible to do in the Graphics mode tha the OS thinks it is in. The OS stores the value of the Graphics mode that it thinks is on the screen in address $57. You can fool the OS into cooperating by POKEing a different value there. POKE the BASIC mode number, not the ANTIC mode number.

The second failure you might get arises when you mix mode lines with different screen memory byte requirements. Some mode lines require 40 bytes per line, some require 20 bytes per line, and some require only 10 bytes per line. Let's say that you insert one 20 byte mode line into a display list with 40 byte mode lines. Then you PRINT text to the display. Everything above the interloper line is fine, but below it the characters are shifted 20 spaces to the right. This is because the operating system assumed that each line would require 40 bytes and positioned the characters accordingly. But ANTIC, when it encountered the interloper line, took only twenty bytes of what the OS thought should be a 40-byte line. ANTIC interpreted the other 20 bytes as belonging to the next line, and displayed them there.

This resulted in the next line and all later lines being shifted 20 spaces to the right.

The only absolute way around this problem is to refrain from using BASIC PRINTs and PLOTs to output to a custom display list screen. The quick and dirty solution is to organize the screen into line groups which contain integer multiples of the standard byte requirement. That is, do not insert a 20-byte mode line into a 40-byte display; instead insert two 20-byte lines or one 20-byte line and two 10-byte lines. So long as you retain the proper integer multiples the horizontal shift will be avoided.

This solution accentuates the third problem with mixed display lists and BASIC: vertical shifts. The OS positions screen material vertically by calculating the number of bytes to skip down from the top of the screen. In a standard 40-byte line display, BASIC would position the characters onto the tenth line by skipping 360 bytes from the beginning. If you have inserted four 10-byte lines then BASIC will end up three lines further down the screen than you would otherwise expect. Furthermore, different mode lines consume different numbers of scan lines, so the position on the screen will not be quite what you expected if you do not take scan line costs into account.

As you can see, mixed mode displays can be difficult to use in conjunction with the OS. Often you must fool the operating system to make such displays work. To PRINT or PLOT to a mode window, POKE the BASIC mode number of that window to address $57, then POKE the address of the top left pixel of the mode window into locations $58 and $59 (lo then hi). In character modes, execute a POSITION 0,0 to home the cursor to the top left corner of the mode window. In map modes, all PLOTs and DRAWTOs will be made using the top left corner of the mode window as the origin of the coordinate system.

The display list system can be used to produce appealing screen displays. Its most obvious use is for mixing text and graphics. For example, you could prepare a screen with a bold BASIC mode 2 title, a medium size BASIC mode 1 subtitle, and small BASIC mode 0 fine print. You could then throw in a BASIC mode 8 picture in the middle with some more text at the bottom. A good example of this technique is provided by the display in the States and Capitals program.

The aforementioned problems will discourage the extensive use of such techniques from BASIC. With assembly language routines modified display lists are best used by organizing the screen into a series of windows, each window having its own LMS instruction and its own independent RAM area.

APPLICATIONS OF DISPLAY LISTS

One simple application of display list modifications is to vertically space lines on the screen by inserting blank line bytes. This will add some vertical spacing which will highlight critical messages and enhance the readability of some displays.

Another important use of display list manipulations is in providing access to features not available from BASIC. There are three text modes supported by ANTIC that BASIC does not support. Only display list manipulations gain the user access to these modes. There are also display list interrupt and fine scrolling capabilities that are only available after the display list is modified. These features are the subjects of chapters 5 and 6.

Manipulations with the LMS instruction and its operand offer many possibilities to the creative programmer. For example, by changing the LMS during vertical blank the programmer can alternate screen images. This can be done at slow speed to change between predrawn displays without having to redraw each one. Each display would continue to reside in (and consume) RAM even while it is not in use, but it would be available almost instantly. This technique can also be used for animation. By flipping through a sequence of displays, cyclic animation can be achieved. The program to do this would manipulate only two address bytes to display many thousands of bytes of RAM.

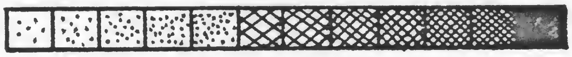
It is also possible to superimpose images by flipping screens at high speed. The human eye has a time resolution of about 1/16th of a second, so a program can cycle between four images, one every 1/60th of a second, so that each repeats every 1/15th of a second. In this way, up to four images can appear to reside simultaneously on the screen. Of course, there are some drawbacks to this method. First, four separate displays may well cost a lot of RAM. Second, each display image will be washed out because it only shows up one quarter of the time. This means that the background of all displays must be black, and each image must be bright. Furthermore, there will be some unpleasant screen flicker when this technique is used. A conservative programmer might consider cycling between only three or even only two images. This technique can also be used to extend the color and luminosity resolution of the computer. By cycling between four versions of the same image, each version stressing one color or luminosity range, a wider range of colors and luminosities is availible. For example, suppose we wish to display a bar of many different luminances. We first set our four color registers to the values:

background:	00
playfield 1:	02
playfield 2:	0A
playfield 3:	0C

Now we put the following images into each of the screen RAM areas:

	pixel contents (by playfield color register)											
first frame	1	1	1	1	2	3	2	3	2	3	2	3
second frame	B	1	1	1	B	B	2	3	2	3	2	3
third frame	B	B	1	1	B	B	B	B	2	3	2	3
fourth frame	B	B	B	1	B	B	B	B	B	B	2	3
effective luminance x4	2	4	6	8	10	12	20	24	30	36	40	48

perceived luminance



In this way much finer luminance resolution is possible.

A final suggestion concerns a subject which is laden with opportunities but little understood as yet: the dynamic display list. This is a display list which the 6502 changes during vertical blank periods. It should be possible to produce interesting effects with dynamic display lists. For example, a text editing program dynamically inserts blank lines above and below the screen line being edited to set it apart from the other lines of text. As the cursor is moved vertically, the display list is changed. The technique is odd but very effective.

CHAPTER 3
GRAPHICS INDIRECTION
(COLOR REGISTERS AND CHARACTER SETS)

Indirection is a powerful concept in computing but a difficult one for the beginning programmer to appreciate. In 6502 assembly language there are three levels of indirection in referring to numbers. The first and most direct level is the immediate addressing mode in which the number itself is directly stated:

```
LDA #$F4
```

The second level of indirection is reached when the program refers to a memory location that holds the number:

```
LDA $0602
```

The third and highest level of indirection with the 6502 is attained when the program refers to a pair of memory locations which together contain the address of the memory location which holds the number. In the 6502, this indirection is complicated by the addition of an index:

```
LDA ($D0),Y
```

Indirection provides a greater degree of generality (and hence power) to the programmer. Instead of trucking out the same old numbers every time she wants to get something done, the programmer can simply point to them. By changing the pointer, she can change the behavior of the program. Indirection is obviously an important capability.

Graphics indirection is built into the ATARI Personal Computer System in two ways: with color registers and with character sets. Programmers first approaching this computer after programming other systems often think in terms of direct colors. A color register is a more complex beast than a color. A color specifies a permanent value. A color register is indirect; it holds any color value. The difference between the two is analogous to the difference between a box-end wrench and a socket wrench. The box-end wrench comes in one size only but a socket wrench can hold almost any size socket. A socket wrench is more flexible but takes a little more skill to utilize properly. Similarly, a color register is more flexible than a color but takes more skill to use effectively.

There are nine color registers in the ATARI 400/800; four are for player-missile graphics and will be discussed in Chapter 4. The remaining five are not always used; depending on the graphics mode used as few as 2 registers or as many as 5 will show up on the screen. In BASIC mode 0 only one and one-half registers are used because the hue value of the characters is ignored; characters take the same hue as playfield register 2 but take their luminosity from register 1. The color registers are in CTIA at addresses $D016 through $D01A. They are 'shadowed' from OS RAM locations into CTIA during vertical blank. Figure 3.1 gives color register shadow and hardware addresses.

IMAGE	HARDWARE		OS SHADOW	
CONTROLLED	LABEL	ADDRESS	LABEL	ADDRESS
player 0	COLPM0	D012	PCOLR0	2C0
player 1	COLPM1	D013	PCOLR1	2C1
player 2	COLPM2	D014	PCOLR2	2C2
player 3	COLPM3	D015	PCOLR3	2C3
playfield 0	COLPF0	D016	COLOR0	2C4
playfield 1	COLPF1	D017	COLOR1	2C5
playfield 2	COLPF2	D018	COLOR2	2C6
playfield 3	COLPF3	D019	COLOR3	2C7
background	COLBK	D01A	COLOR4	2C8

Figure 3.1
color register labels and addresses

For most purposes, the user controls the color registers by writing to the shadow locations. There are only two cases in which the programmer would write directly to the CTIA addresses. The first and most common is the display list interrupt which will be taken up in Chapter 5. The second arises when the user disables the OS vertical blank interrupt routines which move the shadow values into CTIA. Vertical blank interrupts are the subject of Appendix I.

Colors are encoded in a color register by a simple formula. The upper nybble gives the hue value, which is identical to the second parameter of the BASIC SETCOLOR command. Table 9.3 of the BASIC Reference Manual lists hue values. The lower nybble in the color register gives the luminosity value of the color. It is the same as the third parameter in the BASIC SETCOLOR command. The lowest order bit of this nybble is not significant. Thus, there are eight luminosities for each hue. There are a total of 128 colors from which to choose (8 luminosities times 16 hues). In this book, the term 'color' denotes a hue-luminosity combination.

Once a color is encoded into a color register it is mapped onto the screen by referring to the color register that holds it. In map display modes which support four color registers the screen data specifies which color register is to be mapped onto the screen. Since there are four color registers it takes only two bits to encode one pixel. Thus, each screen data byte holds data for four pixels. The value in each pair of bits specifies which color register provides the color for that pixel.

In text display modes (BASIC's GRAPHICS modes 1 and 2) the selection of color registers is made by the top two bits of the character code. This of course leaves only 6 bits for defining the character, which is why these two modes have only 64 characters available.

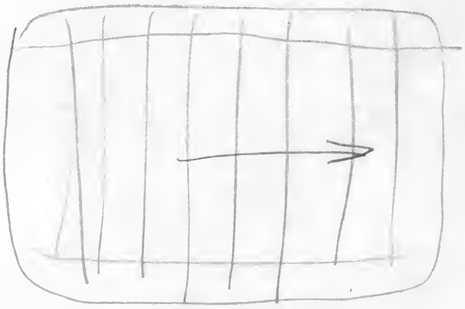
Color register indirection gives the programmer four special capabilities. First, the programmer can choose from 128 different colors for his displays. This allows him to choose the color which most nearly meets his needs.

Second, the programmer can manipulate the color registers in real time to produce pretty effects. The simplest version of this is demonstrated by the following BASIC line:

```
FOR I=0 TO 254 STEP 2:POKE 712,I:NEXTI
```

This line simply cycles the border color through all possible colors. The effect is quite pleasing and certainly grabs attention. The fundamental technique can be extended in a variety of ways. A special variation of this is to create simple cyclic animation by drawing a figure in four colors and then cycling the colors through the color registers rather than redrawing the figure. The following program illustrates the idea:

```
10 GRAPHICS 23
20 FOR X=0 TO 39
30 FOR I=0 TO 3
40 COLOR I
50 PLOT 4*X+I,0
60 DRAWTO 4*X+I,95
70 NEXT I
80 NEXT X
90 A=PEEK(712)
100 POKE 712,PEEK(710)
110 POKE 710,PEEK(709)
120 POKE 709,PEEK(708)
130 POKE 708,A
140 GOTO 90
```



The third application of color registers is to logically key colors to situations. For example, a paged menu system can be made more understandable by changing the background color or the border color for each page in the menu. Perhaps the screen could flash red when an illegal key is pressed. The use of the color characters available in BASIC Graphics modes 1 and 2 can greatly extend the impact of textual material. An account sum could be shown in red if the account is in the red, or black if the account is in the black. Words or phrases of import can be shown in special colors to make them stand out. The use of colors in map modes (no text) can also improve the utility of such graphics. A single graphics image (a monster, a boat, or whatever) could be presented in several different colors to represent several different versions of the same thing. It costs a great deal of RAM to store an image, but it costs very little to change the color of an existing image. For example, it would be much easier to show three different boats by presenting one boat shape in three different colors than three different boat shapes.

The fourth and most important application of color registers is utilized with display list interrupts. A single color register can be used to put up to 128 colors onto a single screen. This very important capability will be discussed in Chapter 5.

CHARACTER SETS

Graphics indirection is also provided through the redefinable character set. A standard character set is provided in ROM, but there is no reason why this particular character set must be used. The user can create and display any character set she desires. There are three steps necessary to use a redefined character set. First, the programmer must define the character set. This is the most time-consuming step. Each character is displayed on the screen on an 8x8 grid; it is encoded in memory as an 8-byte table. Figure 3.2 depicts the encoding arrangement.

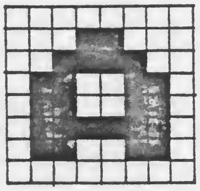
character image	binary representation	hex representation
	00000000	00
	00011000	18
	00111100	3C
	01100110	66
	01100110	66
	01111110	7E
	01100110	66
	00000000	00

Figure 3.2
character encoding

A full character set has 128 characters in it, each with a normal and an inverse video incarnation. Such a character set needs 1024 bytes of space and must start on a 1K boundary. Character sets for BASIC modes 1 and 2 have only 64 distinct characters, and so require only 512 bytes and must start on a 1/2K boundary. The first eight bytes define the zeroth character, the next eight bytes define the first character, and so on. Obviously, defining a new character set is a big job. Fortunately, there are software packages on the market to make this job easier.

Once the character set is defined and placed into RAM, you must tell ANTIC where it can find the character set. This is done by poking the page number of the beginning of the character table into location $D409 (decimal 54281). The OS shadow location, which is the location one would normally use, is called CHBAS and resides at $2F4 (decimal 756). The third step in using character sets is to print the character you want onto the screen. This can be done directly from BASIC with simple PRINTs or by writing numbers directly into the screen memory.

A special capability of the system not supported in BASIC is the 4-color character set option. BASIC Graphics modes 1 and 2 support 5 colors, but each character in these modes is really a two-color character; each one has a foreground color and a background color. The foreground color can be any of four single colors, but only one color at a time can be shown within a single character. This can be a serious hindrance when using character graphics. There are two other text modes designed especially for character graphics. They are ANTIC modes 4 and 5. Each character in these modes is only four pixels wide, but each pixel can have four colors (counting background). The characters are defined just like BASIC Graphics mode 0 characters, except that each pixel is twice as wide and has two bits assigned to it to specify the color register used. Unlike ANTIC modes 6 and 7 (BASIC modes 1 and 2), color register selection is not made by the character name byte but instead by the defined character set. Each byte in the character table is broken into four bit pairs, each of which selects the color for a pixel. (This is why there are only four horizontal pixels per character.) The highest bit (D7) of the character name byte modifies the color register used. Color register selection is made according to Figure 3.3:

bit pair in character defn	D7 = 0	D7 = 1
00	COLBAK	COLBAK
01	PF0	PF0
10	PF1	PF1
11	PF2	PF3

Figure 3.3
color register selection for characters

Using these text modes, multicolored graphics characters can be put onto the screen.

Another interesting ANTIC character mode is the lowercase descenders mode (ANTIC mode 3). This mode displays 10 scan lines per mode line, but since characters use only 8 bytes vertically, the lower two scan lines are normally left empty. If a character in the last quarter of the character set is displayed, the top two scan lines of the character will be left empty; the data that should have been displayed there will instead be shown on the bottom two lines. This allows the user to create lowercase characters with descenders.

Many interesting and useful application possibilities spring from character set indirection. The obvious application is the modified font. A different font can give a program a unique appearance. It is possible to have Greek, Cyrillic, or other special character sets. Going one step further, one can create graphics fonts. The ENERGY CZAR™ computer program uses a redefined character set for bar graphs. A character occupies eight pixels; this means that bar charts implemented with standard characters have a resolution of eight pixels, a rather poor resolution. ENERGY CZAR uses a special character set in which some of the less popular text symbols (ampersands, pound signs, etc) have been replaced with special bar chart characters. One character is a one-pixel bar, another is a two-pixel bar, and so on to the full eight-pixel bar. The program can thus draw detailed bar charts with resolution of a single pixel. Figure 3.4 shows a typical display from this program. The mix of text with map graphics is only apparent; the entire display is constructed with characters.

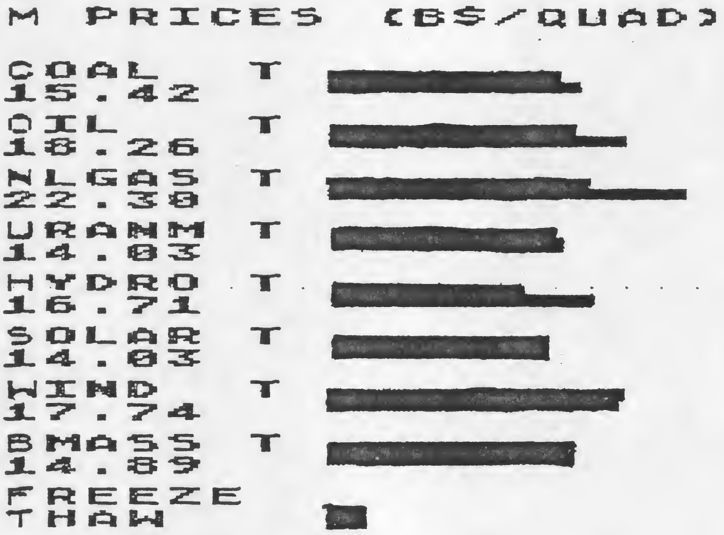
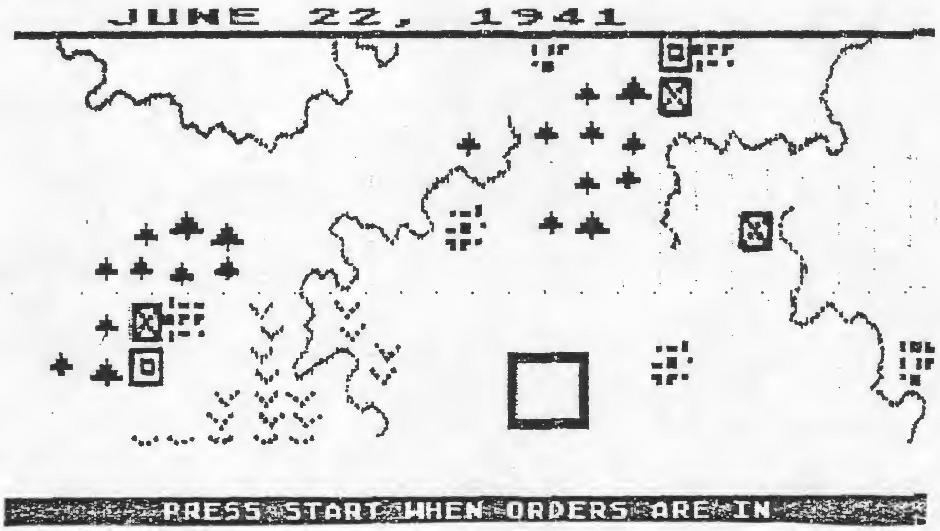


Figure 3.4
ENERGY CZAR™ bar charts

In many applications character sets can be created that show special images. For example, by defining a terrain graphics character set with river characters, forest characters, mountain characters, and so forth, it is possible to make a terrain map of any country. Indeed, with imagination a map of terrain on a different planet can just as easily be done. When doing this, it is best to define five to eight characters for each terrain type. Each variation of a single type should be positioned slightly differently in the character pixel. By mixing the different characters together it is possible to avoid the monotonous look that is characteristic of primitive character graphics. Most people won't realize that the resulting map uses character graphics until they study the map closely. Figure 3.5 shows a display of a terrain map created with character set graphics. The reproduction in black and white does not do justice to the original display, which has up to 18 colors.



JUNE 22, 1941

PRESS START WHEN ORDERS ARE IN

Figure 3.5
terrain map with character set graphics

One could create an electronics character set with transistor characters, diode characters, wire characters, and so forth to produce an electronics schematics program. Or one could create an architectural character set with doorway characters, wall characters, corner characters, and so on to make an architectural blueprint program. The graphics possibilities opened up by character graphics with personal computers have not been fully explored.

Characters can be turned upside down by POKEing a 4 into location 755. One possible application of this feature might be for displaying playing cards (as in a Blackjack game). The upper half of the card can be shown rightside up; with a display list interrupt the characters can be turned upside down for the lower half of the card. This feature might also be of some use in displaying images with mirror reflections (reflection pools, lakes, etc).

Even more exciting possibilities spring to mind when one realizes that it is quite practical to change character sets while the program is running. A character set costs either 512 bytes or 1024 bytes; in either case it is quite inexpensive to keep multiple character sets in memory and flip between them during program execution. There are three time regimes for such character set multiplexing: human slow (more than 1 second); human fast (1/60th second to 1 second); and machine fast (faster than 1/60th second).

Human slow character set multiplexing is useful for 'change of scenery' work. For example, a space travel program might use one graphics character set for one planet, another set for space, and a third set for another planet. As the traveler changes locations, the program changes the character set to give exotic new scenery. An adventure-type program might change character sets as the player changes locales.

Human fast character set multiplexing is primarily of value for animation. This can be done in two ways: changing characters within a single character set, and changing whole character sets. The SPACE INVADERS (trademark of Taito America Corp.) on the ATARI 400/800 uses the former technique. The invaders are actually characters. By rapidly changing the characters, the programmer was able to animate them. This was easy because there are only six different monsters; each has 4 different incarnations. High speed cyclic animation of an entire screen is possible by setting up a number of character sets, drawing the screen image, and then simply cycling through the character sets. If each character has a slightly different incarnation in each of the character sets, that character will go through an animated sequence as the character sets are changed. In this way a screen full of objects could be made to cyclicly move with a very simple loop. Once the character set data is in place and the screen has been drawn the code to animate the screen would be this simple:

```
1000 FOR I=1 TO 10
1010 POKE 756,CHARBASE(I)
1020 NEXT I
1030 GOTO 1000
```

Computer fast character set animation is used to put multiple character sets onto a single screen. This makes use of the display list interrupt capability of the computer. This topic will be addressed further in Chapter 5.

The use of character sets for graphics and animation has many advantages and some limitations. The biggest advantage is that it costs very little RAM to produce detailed displays. A graphics display using BASIC mode 2 characters (such as the one shown in Figure 3.5) can give as much detail and one more color than a BASIC mode 7 display, and yet the character image will cost 200 bytes while the map image will cost 4000 bytes. The RAM cost for multiple character sets is only 512 bytes per set, so it is inexpensive to have multiple character sets. Screen manipulations with character graphics are much faster because you have less data to manipulate. However, character graphics are not as flexible as map graphics. You cannot put anything you want anywhere on the screen. This limitation would preclude the use of character graphics in some applications. However, there remain many graphics applications for which the program need display only a limited number of predefined shapes in fixed locations. In these cases character graphics provide great utility.

CHAPTER 4
PLAYER-MISSILE GRAPHICS

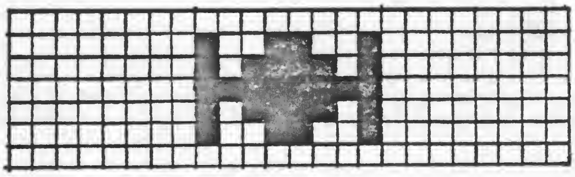
Animation is an important capability of any personal computer system. Activity on the screen can greatly add to the excitement and realism of any program. Certainly animation is crucial to the appeal of many computer games. More important, an animated image can convey information with more impact and clarity than a static image. It can draw attention to an item or event of importance. It can directly show a dynamic process rather than indirectly talk about it. Animation must accordingly be regarded as an important element of the graphics capabilities of any computer system.

The conventional way to effect animation with personal computers is to move the image data through the screen RAM area. This requires a two-step process. First, the program must erase the old image by writing background values to the RAM containing the current image. Then the program must write the image data to the RAM corresponding to the new position of the image. By repeating this process over and over, the image will appear to move on the screen.

There are two problems with this technique. First, if the animation is being done in a graphics mode with large pixels, the motion will not be smooth; the image will jerk across the screen. With other computers the only solution is to use a graphics mode with smaller pixels (higher resolution). The second problem is much worse. The screen is a two-dimensional image but the screen RAM is organized one-dimensionally. This means that an image which is contiguous on the screen will not be contiguous in the RAM. The discrepancy is illustrated in Figure 4.1.

IMAGE

Corresponding bytes in RAM



```
00 00 00
00 99 00
00 BD 00
00 FF 00
00 BD 00
00 99 00
00 00 00
```

spacing of bytes in RAM:

```
00 00 00 00 99 00 00 BD 00 00 FF 00 00 BD 00 00 99 00 00 00 00
```

image bytes scattered through RAM

Figure 4.1
RAM images are not contiguous

The significance of this discrepancy does not become obvious until you try to write a program to move such an image. Look how the bytes that make up the image are scattered through the RAM. To erase them your program must calculate their addresses. This calculation is not always easy to do. The assembly code just to access a single screen byte at screen location (XPOS,YPOS) would look like this (this code assumes 40 bytes per screen line):

```
        LDA SCRNRM    Address of beginning of screen RAM
        STA POINTR    zero page pointer
        LDA SCRNRM+1  high order byte of address
        STA POINTR+1  high order pointer
        LDA #$00
        STA TEMPA+1   temporary register
        LDA YPOS      vertical position
        ASL A         times 2
        ROL TEMPA+1   shift carry into TEMPA+1
        ASL A         times 4
        ROL TEMPA+1   shift carry again
        ASL A         times 8
        ROL TEMPA+1   shift again
        LDX TEMPA+1   save YPOS*8
        STX TEMPB+1   into TEMPB
        STA TEMB      low byte
        ASL A         times 16
        ROL TEMPA+1
        ASL A         times 32
        ROL TEMPA+1
        CLC
        ADC TEMPB     add in YPOS*8 to get YPOS*40
        STA TEMPB
        LDA TEMPA+1   now do high order byte
        ADC TEMPB+1
        STA TEMPB+1
        LDA TEMPB     TEMPB contains the offset from top of screen to pixel
        CLC
        ADC POINTR
        STA POINTR
        LDA TEMPB+1
        ADC POINTR+1
        STA POINTR+1
        LDY XPOS
        LDA (POINTR),Y
```

Clearly, this code to access a screen location is too cumbersome. This is certainly not the most elegant or fastest code to solve the problem; certainly a good programmer could take advantage of special circumstances to make the code more compact or elegant. The point of this is that accessing pixels on a screen takes a lot of computing. The above routine takes about 100 machine cycles to access a single byte on the screen. To move an image that occupies, say, 50 bytes, would require 100 accesses or about 10,000 machine cycles or roughly 10 milliseconds. This may not sound like much but if you want to achieve smooth motion you have to move the object every 17 milliseconds. If there are other objects to move or any calculations to carry out there isn't much processor time left to devote to them. What this all adds up to is that this type of animation (called 'playfield animation') is too slow for many purposes. You can still get animation this way, but you are limited to few objects or small objects or slow motion or few calculations between motion. The trade-offs that a programmer must make in using such animation are too restrictive.

The ATARI 400/800™ solution to this problem is player-missile graphics. In order to understand player-missile graphics, it is important to understand the essence of the problem of playfield animation: the screen image is two-dimensional while the RAM image is one-dimensional. The solution was to create a graphics object which is one-dimensional on the screen as well as one-dimensional in RAM. This object (called a player) appears in RAM as a table that is either 128 or 256 bytes long. The table is mapped directly to the screen. It appears as a vertical band stretching from the top of the screen to the bottom. Each byte in the table is mapped into either one or two horizontal scan lines, with the choice between the two made by the programmer. The screen image is a simple bit-map of the data in the table. If a bit is on, then the corresponding pixel in the vertical column is lit; if the bit is off, then the corresponding pixel is off. Thus, the player image is not strictly one-dimensional; it is actually eight bits wide.

Drawing a player image on the screen is quite simple. First you draw a picture of the desired image on graph paper. This image must be no more than 8 pixels wide. Then you translate the image into binary code, substituting ones for illuminated pixels and zeros for empty ones. Then you translate the resulting binary number into decimal or hexadecimal, depending on which is more convenient Then you store zeros into the player RAM to clear the image. Next, store the image data into the player RAM, with the byte at the top of the player image going first, followed by the other image bytes in top to bottom sequence. The further down in RAM you place the data, the lower the image will appear on the screen.

Animating this image is very easy. Vertical motion is obtained by moving the image data through the player RAM. This is in principle the same method used in playfield animation, but there is a big difference in practice: the move routine for vertical motion is a one-dimensional move instead of a two-dimensional move. The program does not need to multiply by 40 and it often does not need to use indirection. It could be as simple as:

```
     LDX #$01
LOOP LDA PLAYER,X
     STA PLAYER-1,X
     INX
     BNE LOOP
```

This routine takes about 4 milliseconds to move the entire player, about half as long as the playfield animation routine which actually moves only 50 bytes where this one moves 256 bytes. If high speed is necessary, the loop can be trimmed to move only the image bytes themselves rather than the whole player; then the loop would easily run in about 100-200 microseconds. The point here is that vertical motion with players is both simpler and faster than motion with playfield objects.

Horizontal motion is even easier than vertical motion. There is a register for the player called the horizontal position register. The value in this register sets the horizontal position of the player on the screen. All you do is store a number into this register and the player jumps to that horizontal position. To move the player horizontally simply change the number stored in the horizontal position register. That's all there is to it.

Horizontal and vertical motion are independent; you can combine them in any fashion you choose.

The scale for the horizontal position register is one color clock per unit. Thus, adding one to the horizontal position register will move the player one color clock to the right. There are only 228 color clocks in a single scan line; furthermore, some of these are not displayed because of overscan. The horizontal position register can hold 256 positions; some of these are off the left or right edge of the screen. Depending on the overscan of the television, positions 0 through 44 will be off the left edge of the screen and positions 220 through 255 will be off the right edge. Thus, the visible region of the player is in horizontal positions 44 through 220. Remember, however, that this may vary from television to television; a conservative range is from 60 to 200. This coordinate range can sometimes be clumsy to use, but it does offer a nice feature: a simple way to remove a player from the screen is to set the player's horizontal position to zero. With a single load and store in assembly (or a single POKE in BASIC), the player will disappear.

The system described so far makes it possible to produce high-speed animation. There are a number of embellishments which greatly add to its overall utility. The first embellishment is that there are four independent players to use. These players all have their own sets of control registers and RAM area; thus their operation is completely independent. They are labelled P0 through P3. They can be used side by side to give up to 32 bits of horizontal resolution, or they can be used independently to give four movable objects. Each player has its own color register; this color register is completely independent of the playfield color register. The player color registers are called COLP(X) an are shadowed at PCOLR(X). This gives you the capability to put much more color onto the screen. However, each player has only one color; multicolored players are not possible without display list interrupts (display list interrupts are discussed in Chapter 5). Each player has a controllable width; you can set it to have normal width, double width, or quadruple width with the SIZEP(X) registers. This is useful for making players take on different sizes. You also have the option of choosing the vertical resolution of the players. You can use single-line resolution, in which each byte in the player table occupies one horizontal scan line, or double-line resolution, in which each byte occupies two horizontal scan lines. With single-line resolution, each player bit-map table is 256 bytes long; with double-line each table is 128 bytes long. This is the only case where player properties are not independent; the selection of vertical resolution applies to all players. Player vertical resolution is controlled by bit D4 of the DMACTL register. In double-line resolution, the first twn or so bytes in the player table area are lost to vertical overscan and are off the top edge of the screen. The last twenty or so bytes are lost off the bottom edge of the screen. In single-line resolution, twenty and forty bytes are lost correspondingly.

The next embellishment is the provision of missiles. These are two-bit wide graphics objects associated with the players. There is one missile assigned to each player; it takes its color from the player's color register. Missile shape data comes from the missile bit-map table in RAM just in front of the player tables. All four missiles are packed into the same table (four missiles times two bits per missile gives eight bits). Missiles can move independently of players; they have their own horizontal position registers. Missiles have their own size register, SIZEM, which can set the horizontal width just like the SIZEP(X) registers do for players. However, missiles cannot be set to different sizes; they are all set together. Missiles are useful as bullets or for skinny vertical lines on the screen. If desired, the missiles can be grouped together into a fifth player, in which case they take the color of playfield color register 3. This is done by setting bit D4 of the priority control register (PRIOR). Note that missiles can still move independently when this option is in effect; their horizontal positions are set by their horizontal position registers. The fifth player enable bit only affects the color of the missiles.

You move a missile vertically the same way that you move a player: by moving the missile image data through the missile RAM area. This can be difficult to do because missiles are grouped into the same RAM table. To access a single missile you must mask out the bits for the other missiles.

An important feature of player-missile graphics is that players and missiles are completely independent of the playfield. You can mix them with any graphics mode, text or map. This raises a problem: what happens if a player ends up on top of some playfield image? Which image has priority? You have the option to define the priorities used in displaying players. If you wish, all players can have priority over all playfield color registers. Or you can set all playfield color registers (except background) to have priority over all players. Or you can set player 0 and player 1 (henceforth referred to as P0 and P1) to have priority over all playfield color registers, with P2 and P3 having less priority than the playfield. Or you can set playfield color registers 0 and 1 (PF0 and PF1) to have priority over all players, which then have priority over PF2 and PF3. These priorities are selected with the priority control register (PRIOR) which is shadowed at GPRIOR. This capability allows a player to pass in front of of one image and behind another, allowing three-dimensional effects.

The final embellishment is the provision for hardware collision detection. This is primarily of value for games. You can check if any graphics object (player or missile) has collided with anything else.

Specifically, you can check for missile-player collisions, missile-playfield collisions, player-player collisions, and player-playfield collisions. There are 54 possible collisions, and each one has a bit assigned to it that can be checked. If the bit is set, a collision has occurred. These bits are mapped into 15 registers in CTIA (only the lower 4 bits are used and some are not meaningful). These are read-only registers; they cannot be cleared by writing zeros to them. The registers can be cleared for further collision detection by writing any value to register HITCLR. All collision registers are cleared by this command.

In hardware terms, collisions occur when a player image coincides with another image; thus, the collision bit will not be set until the part of the screen showing the collision is drawn. This means that collision detection might not occur until as much as 16 milliseconds have elapsed since the player was moved. The preferred solution is to execute player motion and collision detection during the vertical blank interrupt routine (see Appendix I). In this case collision detection should be checked first, then collisions cleared, then players moved. Another solution is to wait at least 16 milliseconds after moving a player before checking for a collision involving that player.

There are a number of steps necessary to use player-missile graphics. First you must set aside a player-missile RAM area and tell the computer where it is. If you use single-line resolution, this RAM area will be 1280 bytes long; if you use double-line resolution it will be 640 bytes long. A good practice is to use the RAM area just in front of the display area at the top of RAM. The layout of the player-missile area is shown in Figure 4.2.

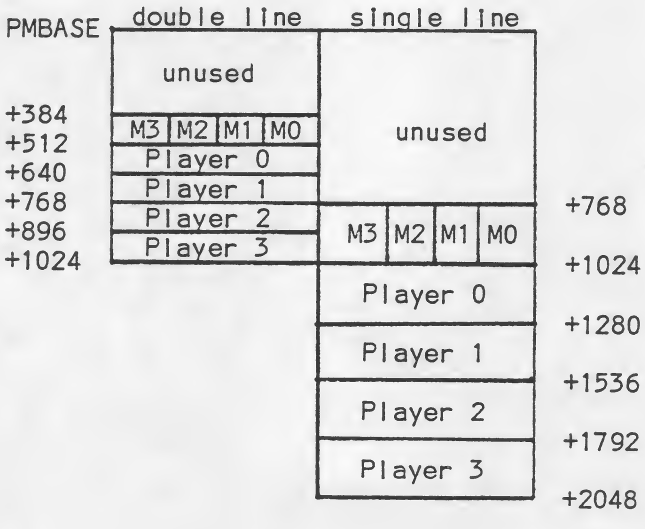


Figure 4.2
player-missile RAM area layout

The pointer to the beginning of the player-missile area is labelled

PMBASE. Because of internal limitations of ANTIC, PMBASE must be on a 1K boundary for single line resolution, or a 2K boundary for double line resolution. If you elect not to use all of the players or none of the missiles, the areas of RAM set aside for the unused objects may be used for other purposes. Once you have decided where your player-missile RAM area will be, you inform ANTIC of this by storing the page number of PMBASE into the PMBASE register in ANTIC.

The next step is to clear the player and missile RAM by storing zeros into all locations in the player-missile RAM area. Then draw the players and missiles by storing image data into the appropriate locations in the player-missile RAM area.

Next, set the player parameters by setting the player color, horizontal position, and width registers to their initial values. If necessary, set the player/playfield priorities. Inform ANTIC of the vertical resolution you desire by setting bit D4 of register DMACTL (shadowed at SDMCTL) for single-line resolution, and clearing the bit for double-line resolution. Finally, enable the players by setting the PM DMA enable bit in DMACTL. Be careful not to disturb the other bits in DMACTL. A sample BASIC program for setting up a player and moving it with the joystick is given below:

```
1   PMBASE=54279:REM                    player-missile base pointer
2   RAMTOP=106:REM                      OS top of RAM pointer
3   SDMCTL=559:REM                      RAM shadow of DMACTL register
4   GRACTL=53277:REM                    CTIA graphics control register
5   HPOSP0=53248:REM                    horizontal position of P0
6   PCOLR0=704:REM                      shadow of player 0 color
10  GRAPHICS 0:SETCOLOR 2,0,0:REM       set background color to black
20  X=100:REM                           BASIC's player horizontal position
30  Y=48:REM                            BASIC's player vertical position
40  A=PEEK(RAMTOP)-8:REM                get RAM 2K below top of RAM
50  POKE PMBASE,A:REM                   tell ANTIC where PM RAM is
60  MYPMBASE=256*A:REM                  keep track of PM RAM address
70  POKE SDMCTL,46:REM                  enable PM DMA with 2-line res
80  POKE GRACTL,3:REM                   enable PM display
90  POKE HPOSP0,100:REM                 declare horizontal position
100 FOR I=MYPMBASE+512 TO MYPMBASE+640:REM this loop clears player
110 POKE I,0
120 NEXT I
130 FOR I=MYPMBASE+512+Y TO MYPMBASE+518+Y
140 READ A:REM                          this loop draws the player
150 POKE I,A
160 NEXT I
170 DATA 8,17,35,255,32,16,8
180 POKE PCOLR0,88:REM                  make the player pink
190 A=STICK(0):REM                      read joystick
200 IF A=15 THEN GOTO 190:REM           if inactive, try again
210 IF A=11 THEN X=X-1:POKE HPOSP0,X
220 IF A=7 THEN X=X+1:POKE HPOSP0,X
230 IF A<>13 THEN GOTO 280
```

```
240 FOR I=8 TO 0 STEP -1
250 POKE MYPMBASE+512+Y+I,PEEK(MYPMBASE+511+Y+I)
260 NEXT I
270 Y=Y+1
280 IF A<>14 THEN GOTO 190
290 FOR I=0 TO 8
300 POKE MYPMBASE+511+Y+I,PEEK(MYPMBASE+512+Y+I))
310 NEXT I
320 Y=Y-1
330 GOTO 190
```

Once players are displayed, they can be difficult to remove from the screen. This is because the procedure by which they are displayed involves several steps. First, ANTIC retrieves player-missile data from RAM (if such retrieval is enabled in DMACTL). Then ANTIC ships the player-missile data to CTIA (if such action is enabled in GRACTL). CTIA displays whatever is in its player and missile graphics registers (GRAFP0 through GRAFP3 and GRAFM). Many programmers attempt to turn off player-missile graphics by clearing the control bits in DMACTL and GRACTL. This only prevents ANTIC from sending new player-missile data to CTIA; the old data in the GRAF(X) registers will still be displayed. To completely clear the players the GRAF(X) registers must be cleared after the control bits in DMACTL and GRACTL have been cleared. A simpler solution is to leave the player up but set its horizontal position to zero. Of course, if this solution is used, ANTIC will continue to use DMA to retrieve player-missile data, wasting roughly 10,000 machine cycles per second.

Player-missile graphics allow a number of very special capabilities. They are obviously of great value in animation. They do have limitations: there are only four players and each is only eight bits wide. If you need more bits of horizontal resolution you can always fall back on playfield animation. But for high speed animation or quick and dirty animation player-missile graphics work very well.

It is possible to bypass ANTIC and write player-missile data directly into the player-missile graphics registers (GRAFP(X)) in CTIA. This gives the programmer more direct control over player-missile graphics. It also increases his responsibilities concomitantly. The programmer must maintain a bit map of player-missile data and move it into the graphics registers at th appropriate times. The 6502 must therefore be slaved to the screen drawing cycle. (See the discussion of kernels in Chapter 5.) This is a clumsy technique that offers minor performance improvements in return for major programming efforts. The programmer who bypasses the hardware power offered by ANTIC must make up for it with his own sweat.

Player-missile graphics offer many capabilities in addition to animation. Players are an excellent way to increase the amount of color in a display. The four additional color registers they provide allow four more colors on each line of the display. Of course, the eight-bit resolution does limit the range of their application. There is a way around this that can sometimes be used. Take a player at quadruple width and put it onto the screen. Then set the priorities so that the player has lower priority than a playfield color. Next reverse that playfield color with background, so that the apparent background color of the screen is really a playfield color. The player disappears behind this new false background. Now cut a hole in the false background by drawing true background on it. The player will show up in front of the true background color, but only in the area where true background has been drawn. In this way the player can have more than eight bits of horizontal resolution. A sample program for doing this is:

```
1  RAMTOP=106:REM                      OS top of RAM pointer
2  PMBASE=54279:REM                    ANTIC player-missile RAM pointer
3  SDMCTL=559:REM                      shadow of DMACTL
4  GRACTL=53277:REM                    CTIA graphics control register
5  HPOSP0=53248:REM                    horizontal position register of P0
6  PCOLR0=704:REM                      shadow of player 0 color register
7  SIZEP0=53256:REM                    player width control register
8  GPRIOR=623:REM                      priority control register
10 GRAPHICS 7
20 SETCOLOR 4,8,4
30 SETCOLOR 2,0,0
40 COLOR 3
50 FOR Y=0 TO 79:REM                   this loop fills the screen
60 PLOT 0,Y
70 DRAWTO 159,Y
80 NEXT Y
90 A=PEEK(RAMTOP)-20:REM               must back up further for GR. 7
100 POKE PMBASE,A
110 MYPMBASE=256*A
120 POKE SDMCTL,46
130 POKE GRACTL,3
140 POKE HPOSP0,100
150 FOR I=MYPMBASE+512 TO MYPMBASE+640
160 POKE I,255:REM                     make player solid color
170 NEXT I
180 POKE PCOLR0,88
190 POKE SIZEP0,3:REM                  set player to quadruple width
200 POKE GPRIOR,4:REM                  set priority
210 COLOR 4
220 FOR Y=30 TO 40
230 PLOT Y+22,Y
240 DRAWTO Y+43,Y
250 NEXT Y
```

This program produces the following display:

Figure 4.3
masking a player for more resolution

Another application of player-missile graphics is for special characters. There are many special types of characters that cross vertical boundaries in normal character sets. One way to deal with these is to create special character sets that address this problem. Another way is to use a player. Subscripts, integral signs, and other special symbols can be done this way. A sample program for doing this is:

```
1  RAMTOP=106:REM                      OS top of RAM pointer
2  PMBASE=54279:REM                    ANTIC player-missile RAM pointer
3  SDMCTL=559:REM                      shadow of DMACTL
4  GRACTL=53277:REM                    CTIA's graphics control register
5  HPOSP0=53248:REM                    horizontal position register of P0
6  PCOLR0=704:REM                      shadow of player 0 color register
10 GRAPHICS 0:A=PEEK(RAMTOP)-16:REM    must back up for 1-line resolution
20 POKE PMBASE,A
30 MYPMBASE=256*A
40 POKE SDMCTL,62
50 POKE GRACTL,3
60 POKE HPOSP0,102
70 FOR I=MYPMBASE+1024 TO MYPMBASE+1280
80 POKE I,0
90 NEXT I
100 POKE PCOLR0,140
110 FOR I=0 TO 15
120 READ X
130 POKE MYPMBASE+1100+I,X
140 NEXT I
150 DATA 14,29,24,24,24,24,24,24
160 DATA 24,24,24,24,24,24,184,112
170 ?" ":REM                           clear screen
180 POSITION 15,6
190 ?"xdx"
```

This program produces the following display:

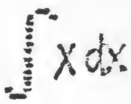


Figure 4.4
using a player as a special character

A particularly useful application of players is for cursors. With their ability to smoothly move anywhere over the screen without disturbing its contents they are ideally suited for such applications. The cursor can change color as it moves over the screen to indicate what it has under it.

Player-missile graphics provide many capabilities. Their uses for action games as animated objects are obvious. They have many serious uses as well. They can add color and resolution to any display. They can present special characters. They can be used as cursors. Use them.

CHAPTER 5
DISPLAY LIST INTERRUPTS

The display list interrupt is one of the most powerful capabilities built into the ATARI Personal Computer System. It is also one of the least accessible features of the system, requiring a firm understanding of assembly language as well as all of the other characteristics of the machine. Display list interrupts all by themselves provide no additional capabilities; they must be used in conjunction with the other features of the system such as player-missile graphics, character set indirection, or color register indirection. With display list interrupts the full power of these features can be deployed.

Display list interrupts take advantage of the sequential nature of the raster scan television display. The television draws the screen image in a time sequence. It draws images from the top of the screen to the bottom. This drawing process takes about 13,000 microseconds, which looks instantaneous to the human eye, but is a long time in the time scale that the computer works in. The computer has plenty of time to change the parameters of the screen display while it is being drawn. Of course, it must effect each change each time the screen is drawn, which is 60 times per second. Also (and this is the tricky part), it must change the parameter in question at exactly the same time each time the screen is drawn. That is, the cycle of changing screen parameters must be synchronized to the screen drawing cycle. One way to do this might be to lock the 6502 up into a tight timing loop whose execution frequency is exactly 60 Hertz. This would make it very difficult to carry out any computations other than the screen display computations. It would also be a tedious job. A much better way would be to interrupt the 6502 just before the time has come to change the screen parameters. The 6502 responds to the interrupt, changes the screen parameters, and returns to its normal business. The interrupt to do this must be precisely timed to occur at exactly the same time during the screen drawing process. This specially timed interrupt is provided by the ANTIC chip; it is called a display list interrupt (DLI).

The timing and execution of any interrupt process can be intricate; therefore I shall first narrate the sequence of events in a properly working display list interrupt. The process begins when the ANTIC chip encounters a display list instruction with its interrupt bit (bit D7) set. ANTIC waits until the last scan line of the mode line it is currently displaying. ANTIC then refers to its NMIEN bit 7 $D40E register to see if display list interrupts have been enabled. If the enable bit is clear, ANTIC ignores the interrupt and continues its regular tasks. If the enable bit is set, ANTIC pulls down the NMI line on the 6502. ANTIC then goes back to its normal display activities. The 6502 vectors through the NMI vector to an interrupt service routine in the OS. This routine first determines the cause of the interrupt. If the interrupt is indeed a display list interrupt, the routine vectors through addresses $0200, $0201 (lo then hi) to a DLI service routine. The DLI routine changes one or more of the graphics registers which control the display. Then the 6502 RTIs to resume its mainline program.

There are a number of steps involved in setting up a DLI. The very first thing you must do is write the DLI routine itself. The routine must push any 6502 registers that will be altered onto the stack, as the OS interrupt poll routine saves no registers. (The 6502 does automatically push the Processor Status Register onto the stack.) The routine should be short and fast; it should only change registers related to the display. It should end by restoring any 6502 registers pushed onto the stack. Next you must place the DLI service routine somewhere in memory. Page 6 is an ideal place. Set the vector at $0200, $0201 to point to your routine. Determine the vertical point on the screen where you want the DLI to occur, then go to the corresponding display list instruction and set bit D7 of the previous instruction. Finally, enable the DLI by setting bit D7 of the NMIEN register at $D40E. The DLI will immediately begin functioning.

As with any interrupt service routine, timing considerations can be critical. ANTIC does not send the interrupt to the 6502 immediately upon encountering an interrupt instruction; it delays this until the last scan line of the interrupting mode line. The 6502 and the interrupt service routine in the OS together consume between 18 and 25 machine cycles. Thus, the very first instruction of your DLI service routine will not be reached until at least 18 machine cycles have elapsed in the last scan line of the interrupting mode line. 18 machine cycles correspond to 36 color clocks on the screen. Thus, your DLI service routine will begin executing while the electron beam is partway across the screen in the last scan line of the interrupting mode line. For example, if such a DLI routine changes a color register, the old color will be displayed on the left half of the screen and the new color will show up on the right half of the screen. Because of uncertain timing in the response of the 6502 to an interrupt, the border between them will not be sharp but will jiggle back and forth irritatingly.

There is a solution to this problem. It is provided in the form of the WSYNC (wait for horizontal sync) register. Whenever this register is addressed in any way, the ANTIC chip pulls down the RDY line on the 6502. This effectively freezes the 6502 until the register is reset by a horizontal sync. The effect is that the 6502 freezes until the electron beam returns to the left edge of the screen. If you insert a STA WSYNC instruction just before the instruction which stores a value into a color register, the color will go in while the beam is off the left edge of the screen. The color transition will occur one scan line lower, but will be neat and clean.

The proper use of a DLI then is to set the DLI bit on the mode line BEFORE the mode line for which you want the action to occur. The DLI service routine should first save the 6502 registers onto the stack, and then load the 6502 registers with the new graphics values to be used. It should execute a STA WSYNC, and then store the new values into the appropriate ANTIC or CTIA registers. Finally, it should restore the 6502 registers and return from the interrupt. This procedure will guarantee that the graphics registers are changed at the beginning of the desired line while the electron beam is off the screen.

A simple program demonstrating a DLI is given below:

```
10 DLIST=PEEK(560)+256*PEEK(561):REM   find display list
20 POKE DLIST+15,130:REM               insert interrupt instruction
30 FOR I=0 TO 19:REM                   loop for poking DLI service routine
40 READ A:POKE 1536+I,A:NEXT I
50 DATA 72,138,72,169,80,162,88
60 DATA 141,10,212,141,23,208
70 DATA 141,24,208,104,170,104,64
80 POKE 512,0:POKE 513,6:REM           poke in interrupt vector
90 POKE 54286,192:REM                  enable DLI
```

This routine uses the following assembly language DLI service routine:

```
PHA          save accumulator
TXA
PHA          save X-register
LDA #$50     dark color for characters
LDX #$58     pink
STA WSYNC    wait
STA COLPF1   store color
STX COLPF2   store color
PLA
TAX
PLA          restore registers
RTI          done
```

NMIEN

NMIST

NMIRES

This is a very simple DLI routine. It changes the background color from blue to pink. It also changes the color of the characters so that they show up as dark against the pink background. One might wonder why the upper half of the screen remains blue even though the DLI routine keeps stuffing pink into the color register. The answer is that the OS vertical blank interrupt routine keeps stuffing blue into the color register during the vertical blank period. The blue color comes from the OS shadow register for that color register. Every hardware color register is shadowed out to a RAM location. You may already know about these shadow registers at locations 708 through 712. For most purposes you can change colors by poking values into the shadow registers. If you poke directly into the hardware registers, the OS shadow process will wipe out your poked color within a 60th of a second. For DLI'S, however, you must store your new color values directly into the hardware registers. You can not use a DLI to set the color of the first displayed line of the screen; the OS takes care of that line for you. Use DLI's to change colors of lines below the first line.

By stuffing colors directly into the hardware registers you create a new problem: you defeat the automatic attract mode. Attract mode is a feature provided by the operating system. After nine minutes without a keypress, the colors on the screen begin to cycle through random hues at lowered luminances. This insures that a computer left unattended for several hours does not burn an image into the television screen. It is easy to build attract mode into a display list interrupt. Only two lines of assembly code need be inserted into the DLI routine:

```
  OLD                 NEW

LDA NEWCOL          LDA NEWCOL
STA WSYNC           EOR COLRSH 4F
STA COLPF2          AND DRKMSK 4E
                    STA WSYNC
                    STA COLPF2
```

DRKMSK and COLRSH are zero page locations ($4E and $4F) set up and updated by the OS during vertical blank interrupt. When attract mode is not in force COLRSH takes a value of 00 and DRKMSK takes $FF. When attract mode is in force, COLRSH is given a new random value every four seconds and DRKMSK holds a value of $F6. Thus, COLRSH scrambles the color and DRKMSK lops off the highest luminance bit.

The implementation of attract mode in DLI's exacerbates an already difficult problem: the shortage of execution time during a DLI. A description of DLI timing will make the problem more obvious. DLI execution is broken into three phases. Phase One covers the period from the beginning of the DLI to the STA WSYNC instruction. During Phase One the electron beam is drawing the last scan line of the interrupting mode line. Phase Two covers the period from the STA WSYNC instruction to the appearance of the beam on the television screen. Phase Two corresponds to horizontal blank; all graphics changes should be made during Phase Two. Phase Three covers the period from the appearance of the beam on the screen to the end of the DLI service routine. The timing of Phase Three is not critical.

One horizontal scan line takes 114 clock cycles of real time. A DLI reaches the 6502 on or around cycle number 15. The 6502 takes about 7 cycles to respond to the interrupt. The OS routine to service the interrupt and vector it on to the DLI service routine takes 11 machine cycles. Thus, the DLI service routine is not reached until about 33 clock cycles have elapsed. Furthermore, the STA WSYNC instruction must be begun by cycle number 103; this reduces the time available in Phase One by 11 cycles. Finally, ANTIC's DMA will steal some of the remaining clock cycles from the 6502. Nine cycles will be lost to memory refresh DMA. This leaves an absolute maximum of 61 cycles available for Phase One. This maximum is achieved only with blank line mode lines. Character and map mode instructions will result in the loss of one cycle for each byte of display data. The worst case arises with BASIC modes 0, 7, and 8, which require 40 bytes per line. Only 21 machine cycles are available to Phase One in such modes. Thus, a Phase One routine will have from 21 to 61 machine cycles of execution time available to it.

Phase Two, the critical phase, extends over 24 clock cycles of real time. As with Phase One, some of these cycles are lost to cycle stealing DMA. Player-missile graphics will cost five cycles if they are used. The display instruction will cost one cycle; if the LMS option is used two more cycles will be stolen. Finally, one or two cycles may be lost to memory refresh or display data retrieval. Thus, from 14 to 23 machine cycles are available to Phase Two.

The problems of DLI timing now become obvious. To load, attract and store a single color will consume 14 cycles. Saving A, X, and Y onto the stack and then loading, attracting, and saving 3 colors into A, X, and Y will cost 47 cycles, most if not all of Phase One. Obviously, the programmer who wishes to use DLI's for extensive graphics changes will expend much effort on the timing of the DLI. Fortunately, the beginning programmer need not concern himself with extensive timing calculations. If only single color changes or simple graphics operations are to be performed, cycle counting and speed optimization are unnecessary. These considerations are only important for high-performance situations.

There are no simple options for the programmer who needs to change more than three color registers in a single DLI. It might be possible to load, attract, and store a fourth color early in Phase Three if that color is not displayed on the left edge of the screen. Similarly, a color not showing up on the right side of the screen could be changed during Phase One. Another approach is to break one overactive DLI into two less ambitious DLI's, each doing half the work of the original. The second DLI could be provided by inserting a single scan line blank instruction (with DLI bit set) into the display list just below the main interrupting mode line. This will of course consume some screen space.

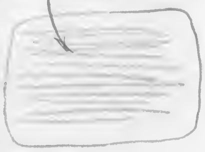
Another partial solution is to perform the attract chores during vertical blank periods. To do this, two tables of colors must be kept in RAM. The first table contains color values intended to be displayed by the DLI routines. The second table contains the attracted values of these colors. During vertical blank, a user-supplied interrupt service routine fetches each color from the first table, attracts it, and stores the attracted color to the second table. The DLI routine then retrieves values directly from the second table without paying the time penalty for attract.

It is often desirable to have a number of DLI's occurring at several vertical positions on the screen. This is an important way to add color to a display. Unfortunately, there is only one DLI vector; if multiple DLI's are to be implemented then the vectoring to the appropriate DLI must be implemented in the DLI routine itself. There are several ways to do this. If the DLI routine does the same process with different values then it can be table-driven. On each pass through the DLI routine a counter is incremented and used as an index to a table of values. A sample DLI routine for doing this is as follows:

```
        PHA
        TXA
        PHA
        INC COUNTR
        LDX COUNTR
        LDA COLTAB,X    use page $F0  for color table
        STA WSYNC       wait
        STA COLBAK
        CPX #$4F        last line?
        BNE ENDDLI      no, exit
        LDA #$00        yes, reset counter
        STA COUNTR
ENDDLI  PLA
        TAX
        PLA             restore accumulator
        RTI
```

The BASIC program to call this routine is:

```
10 GRAPHICS 7
20 DLIST=PEEK(560)+256*PEEK(561):REM   find display list
30 FOR J=6 TO 84:REM                   give every mode line a DLI
40 POKE DLIST+J,141:REM                BASIC mode 7 with DLI bit set
50 NEXT J
60 FOR J=0 TO 30
70 READ A:POKE 1536+J,A:NEXTJ:REM      poke in DLI service routine
80 DATA 72,138,72,238,32,6,175,32,6
90 DATA 189,0,240,141,10,212,141,26,208
100 DATA 224,79,208,5,169,0
110 DATA 141,32,6,104,170,104,64
120 POKE 512,0:POKE 513,6:REM          vector to DLI service routine
130 POKE 54286,192:REM                 enable DLI
```



80 lines of color jiggles w/ keypress

This program will put 80 different colors onto the screen.

There are other ways to implement multiple DLI's. One way is to use a DLI counter as a test for branching through the DLI service routines to the proper DLI service routine. This slows down the response of all the DLI's, particularly the ones at the end of the test sequence. Another way is to have each DLI service routine write the address of the next routine into the DLI vector at $200, $201. This should be done during Phase Three. This is the most general solution to the problem of multiple DLI's. It has the additional advantage that vectoring logic is performed after the time critical portion of the DLI, not before.

The OS keyboard click routine interferes with the function of the DLI. Whenever a key is pressed and acknowledged, the onboard speaker is clicked. The timing for this click is provided by several STA WSYNC instructions.

This can throw off the timing of a DLI routine and cause the screen colors to jump downward by one scan line for a fraction of a second. There is no easy solution to this problem. One possible solution involves the VCOUNT register, a read-only register in ANTIC which tells what scan line ANTIC is displaying. A DLI routine could examine this register to decide when to change a color. Another solution is to disable the OS keyboard service routine and provide your own keyboard routine. This would be a tedious job. The final solution is to accept no inputs from the keyboard. If keypresses are not acknowledged, the screen jiggle does not occur.

The DLI was designed to replace a more primitive software/hardware technique called a kernel. A kernel is a 6502 program loop which is precisely timed to the display cycle of the television set. By monitoring the VCOUNT register and consulting a table of screen changes catalogued as a function of VCOUNT values, the 6502 can arbitrarily control all graphics values for the entire screen. A high price is paid for this power: the 6502 is not available for computations during the screen display time, which is about 75% of the time. Furthermore, no computation may consume more than the 4000 or so machine cycles available during vertical blank and overscan periods. This restriction means that kernels can only be used with programs requiring little computation, such as certain skill and action games. For example, the BASKETBALL program for the ATARI 400/800™ uses a kernel; the program requires little computation but much color. The multi-colored players in this game could not be done with display list interrupts, because DLI's are keyed to playfield vertical positions, not player positions.

It is possible to extend the kernel idea right into a single scan line and change graphics registers on the fly. In this way a single color register can present several colors on a single scan line. The horizontal position of the color change is determined by the amount of time that elapses before the change goes in. Thus, by carefully counting machine cycles the programmer can get more graphics onto the screen. Unfortunately, this is extremely difficult to achieve in practice. With ANTIC DMAing the 6502, it is very difficult to know exactly how many cycles have really elapsed; a simple count of 6502 cycles is not adequate. If ANTIC's DMA is turned off, the 6502 can assume full control of the display but must then perform all the work that ANTIC normally does. For these reasons horizontal kernels are seldom worth the effort. However, if the two images to be displayed in different colors are widely separated, say by 20 color clocks or more, the separation should cover up the timing uncertainties and render this technique feasible.

The tremendous value of graphics indirection and all those modifiable registers in the hardware now becomes obvious. With display list interrupts, every one of those registers can be changed on the fly. You can put lots of color, graphics, and special effects onto the screen. The most obvious application of DLI's is to put more color onto the screen. Each color register can be changed as many times as you have DLI's. This applies to both playfield color registers and player color registers. Thus, you have up to nine color registers, each of which can display up to 128 different colors. Is that enough color for you? Of course, a normal program would not lend itself to effectively using all of those colors. Too many DLI's start slowing down the whole program. Sometimes the screen layout cannot accomodate lots of DLI's. In practice, a dozen colors is easy, two dozen requires careful planning, and more than that requires a contrived situation.

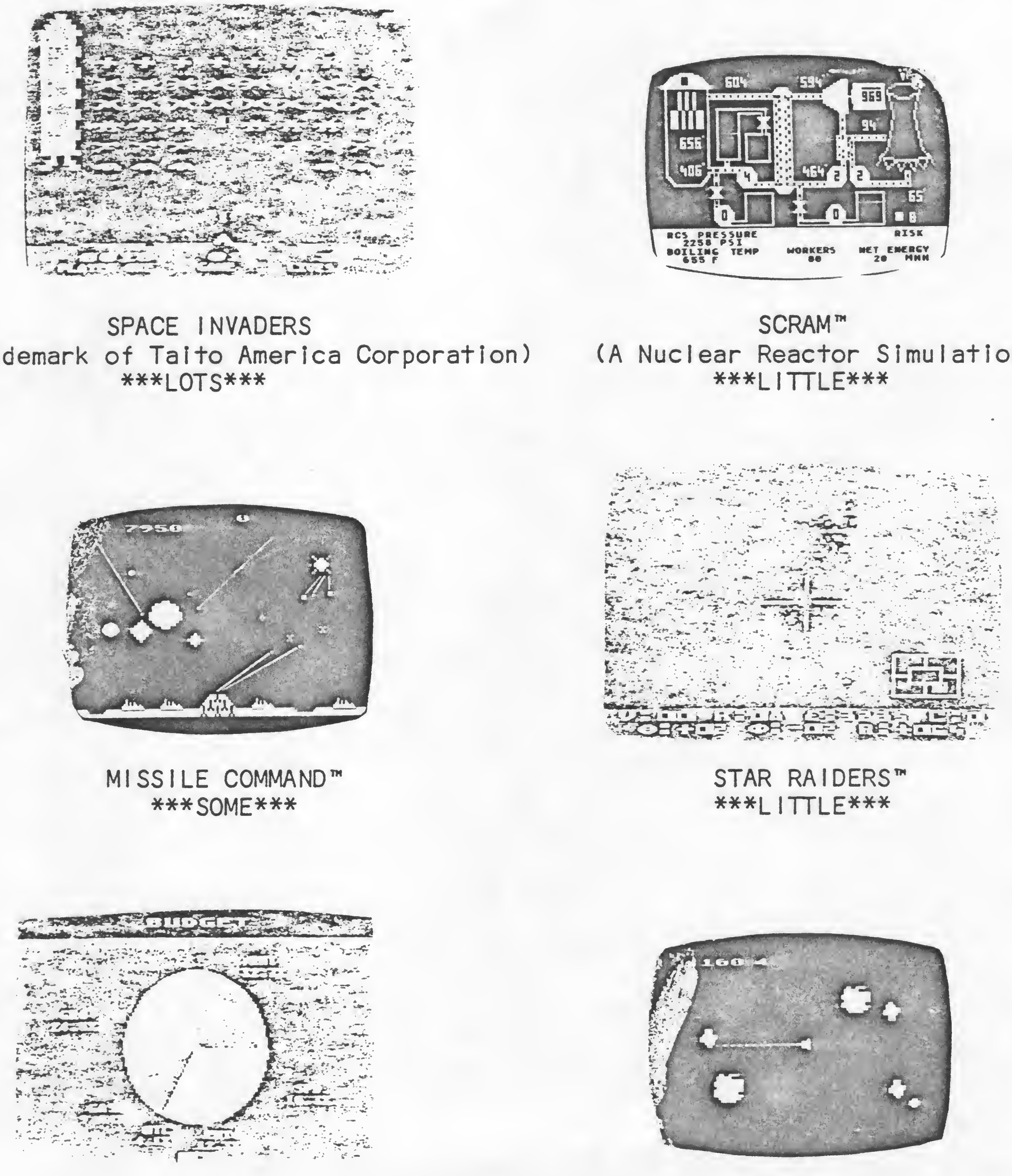
DLI's can give more than color; they can also be used to extend the power of player-missile graphics. The horizontal position of a player can be changed by a DLI. In this way a player can be repositioned partway down the screen. A single player can have several incarnations on the screen. If you imagine a player as a vertical column with images drawn on it, a DLI becomes a pair of scissors with which you can snip the column and reposition sections of it on the screen. Of course, no two sections of the player can be on the same horizontal line, so two incarnations of the player cannot be on the same horizontal line. If your display needs allow graphics objects that will never be on the same horizontal line, a single player can do the job.

Another way DLI's can be used in conjunction with players is to change their width or priority. This would most often be used along with the priority masking trick described in Chapter 4.

The last application of DLI's is the changing of character sets partway down the screen. This allows a program to use character graphics in a large window and regular text in a text window. Multiple character set changes are possible; a program might use one graphics character set at the top of the screen, another graphics character set in the middle of the screen, and a regular text character set at the bottom. A 'Rosetta Stone' program would also be possible, showing different text fonts on the same screen. The vertical reflect bit can be changed with a DLI routine, allowing some text to be rightside up and other text to be upside down.

The proper use of the DLI requires careful layout of the screen display. The designer must give close consideration to the vertical architecture of her display. The raster scan television system is not two-dimensionally symmetric; it has far more vertical structure than horizontal structure. This is because the pace for horizontal screen drawing is about 200 times faster than the pace for vertical screen drawing. The ATARI 400/800™ display system was designed specifically for raster scan television, and it mirrors the anisotropy of the raster scan system. The ATARI 400/800™ display is not a flat, blank sheet of paper on which you draw; it is a stack of thin strips, each of which can take different parameters. The programmer who insists on designing an isotropic display wastes many opportunities. You will achieve optimal results when you organize the information you wish to display in a strong vertical structure. This allows the full power of the DLI to be brought to bear.

Figure 5.1 shows some screen displays from various programs and gives estimates of the degree of vertical screen architecture used in each.



SPACE INVADERS
(Trademark of Taito America Corporation)
LOTS

SCRAM™
(A Nuclear Reactor Simulation)
LITTLE

MISSILE COMMAND™
SOME

STAR RAIDERS™
LITTLE

GRAPH IT™
NONE

ASTEROIDS™
NONE

Figure 5.1
examples of vertical screen architecture

CHAPTER 6
SCROLLING

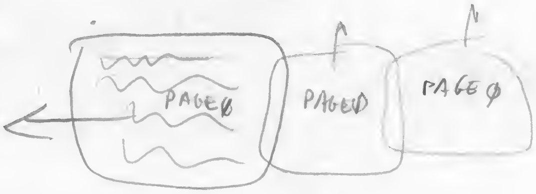
Quite frequently the amount of information that a programmer wants to display exceeds the amount of information that can fit onto the screen. One way of solving this problem is to scroll the information across the display. For example, listings of BASIC programs scroll vertically from the bottom to the top of the screen. All personal computers implement this type of scrolling. However, the ATARI Personal Computer System has two additional scrolling facilities that offer exciting possibilities. The first is 'Load Memory Scan' (LMS) coarse scrolling; the second is fine scrolling.

Conventional computers use coarse scrolling; in this type of scrolling the pixels that hold the characters are fixed in position on the screen and text is scrolled by moving bytes through the screen RAM. The resolution of the scrolling is a single character pixel, which is very coarse. The scrolling this produces is jerky and quite unpleasant. Furthermore, it is achieved by moving up to a thousand bytes around in memory, a slow and clumsy task. In essence, the program must move data through the playfield to scroll.

Some personal computers can produce a somewhat finer scroll by drawing images in a higher resolution graphics mode and then scrolling these images. Although higher scrolling resolution is achieved, more data must be moved to attain the scrolling and the program is consequently slowed. The fundamental problem is that the scrolling is implemented by moving data through the screen area.

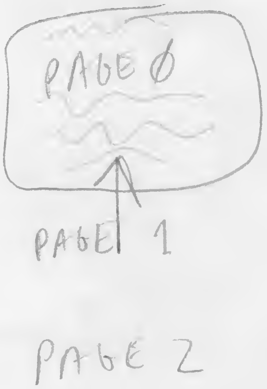
There is a better way to achieve coarse scrolling with the ATARI 400/800™: move the screen area over the data. The display list opcodes support a feature called Load Memory Scan. The LMS instruction was first described in Chapter 2. The LMS instruction tells ANTIC where the screen memory is. A normal display list will have one LMS instruction at the beginning of the display list; the RAM area it points to provides the screen data for the entire screen in a linear sequence. By manipulating the operand bytes of the LMS instruction, a primitive scroll can be implemented. In effect, this moves the playfield window over the screen data. Thus, by manipulating just two address bytes, you can produce an effect identical to moving the entire screen RAM. The following program does just that:

```
10 DLIST=PEEK(560)+256*PEEK(561):REM   find display list
20 LMSLOW=DLIST+4:REM                  get low address of LMS operand
30 LMSHIGH=DLIST+5:REM                 get high address of LMS operand
40 FOR I=0 TO 255:REM                  outer loop
50 POKE LMSHIGH,I
60 FOR J=0 TO 255:REM                  inner loop
70 POKE LMSLOW,J
80 FOR Y=1 TO 50:NEXT Y:REM            delay loop
90 NEXT J
100 NEXT I
```



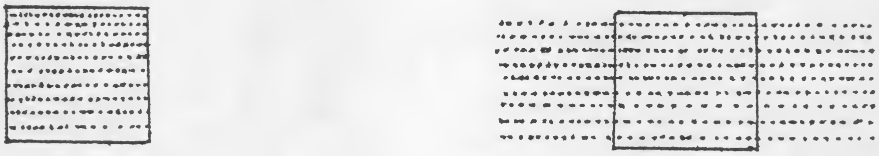
This program sweeps the display over the entire address space of the computer. The contents of the memory are all dumped onto the screen. The scroll is a clumsy serial scroll combining horizontal scrolling with vertical scrolling. A pure vertical scroll can be achieved by adding or subtracting a fixed amount (the line length in bytes) to the LMS operand. The following program does that:

```
10 GRAPHICS 0
20 DLIST=PEEK(560)+256*PEEK(561)
30 LMSLOW=DLIST+4
40 LMSHIGH=DLIST+5
50 SCREENLOW=0
60 SCREENHIGH=0
70 SCREENLOW=SCREENLOW+40:REM        next line
80 IF SCREENLOW<256 THEN GOTO 120:REM  overflow?
90 SCREENLOW=SCREENLOW-256:REM       yes, adjust pointer
100 SCREENHIGH=SCREENHIGH+1
110 IF SCREENHIGH=256 THEN END
120 POKE LMSLOW,SCREENLOW
130 POKE LMSHIGH,SCREENHIGH
140 GOTO 70
```



A pure horizontal scroll is not so simple to do as a pure vertical scroll. The problem is that the screen RAM for a simple display list is organized serially. The screen data bytes for the lines are strung in sequence, with the bytes for one line immediately following the bytes for the previous line. We can horizontally scroll the lines by shifting all the bytes to the left; this is done by decrementing the LMS operand. However, the leftmost byte on each line will then be scrolled into the rightmost position in the next higher line. The first sample program illustrated this problem.

The solution is to expand the screen data area and break it up into a series of independent horizontal line data areas. Figure 6.1 schematically illustrates this idea:



normal data arrangement

arrangement for horizontal scroll

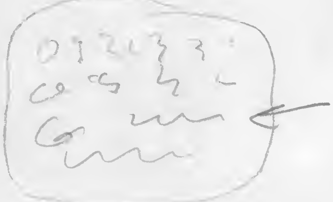
Figure 6.1
arranging screen RAM

On the left is the normal arrangement. One-dimensional serial RAM is stacked in linear sequence to create the screen data area. On the right is the arrangement we need for proper horizontal scrolling. The RAM is of course still one-dimensional and still serial, but now it is used differently. The RAM for each horizontal line extends much further than the screen can show. This is no accident; the whole point of scrolling is to allow a program to display more information than the screen can hold. We can't show all that extra information if we don't allocate the RAM to hold it. With this arrangement we can implement true horizontal scrolling. We can move the screen window over the screen data without the undesirable vertical roll of the earlier approach.

The first step in implementing pure horizontal scroll is to determine the total horizontal line length and allocate RAM accordingly. Next, we must write a completely new display list with an LMS instruction on each mode line. The display list will of course be longer than usual, but there is no reason why we cannot write such a display list. What values do we use for the LMS operands? It is most convenient to use the address of the first byte of each horizontal screen data line, the points marked with x's on the diagram. There will be one such address for each mode line on the screen. Once the new display list is in place, ANTIC must be turned onto it and screen data must be written to populate the screen. To execute a scroll, each and every LMS operand in the display list must be incremented for a rightward scroll or decremented for a leftward scroll. Program logic must insure that the image does not scroll beyond the limits of the allocated RAM areas; otherwise, garbage displays will result. In setting up such logic, the programmer must remember that the LMS operand points to the first screen data byte in the displayed line. The maximum value of the LMS operand is equal to the address of the last byte in the long horizontal line minus the number of bytes in one displayed line.

As this process is rather intricate, let us work out an example. First, we must select our total horizontal line length. We shall use a horizontal line length of 256 bytes, as this will simplify address calculations. Each horizontal line will then require one page of RAM. Since we will use BASIC mode 2, there will be 12 mode lines on screen; thus, 12 pages or 3K of RAM will be required. For simplicity (and to guarantee that our screen RAM will be populated with nonzero data), we will use the bottom 3K of RAM. This area is used by the OS and DOS and so should be full of interesting data. To make matters more interesting, we'll put the display list onto page 6 so that we can display the display list on the screen as we are scrolling. The initial values of the LMS operands will thus be particularly easy to calculate; the low order bytes will all be zeros and the high order bytes will be (in order) 0, 1, 2, etc. The following program performs all these operations and scrolls the screen horizontally:

```
10 REM first set up the display list
20 POKE 1536,112:REM                     8 blank lines
30 POKE 1537,112:REM                     8 blank lines
40 POKE 1538,112:REM                     8 blank lines
50 FOR I=1 TO 12:REM                     loop to put in display list
60 POKE 1536+3*I,71:REM                  BASIC mode 2 with LMS set
70 POKE 1536+3*I+1,0:REM                 low byte of LMS operand
80 POKE 1536+3*I+2,I:REM                 high byte of LMS operand
90 NEXT I
100 POKE 1575,65:REM                     ANTIC JVB instruction
110 POKE 1576,0:REM                      display list starts at $0600
120 POKE 1577,6
130 REM tell ANTIC where display list is
140 POKE 560,0
150 POKE 561,6
160 REM now scroll horizontally
170 FOR I=0 TO 235:REM                   loop through LMS low bytes
175 REM we use 235---not 255---because screen width is 20 characters
180 FOR J=1 TO 12:REM                    for each mode line
190 POKE 1536+3*J+1,I:REM                put in new LMS low byte
200 NEXT J
210 NEXT I
220 GOTO 170:REM                         endless loop
```



large characters

This program scrolls the data from right to left. When the end of a page is reached it simply starts over at the beginning. The display list can be found on the sixth line down (it's on page six). It appears as a sequence of double quotation marks.

The next step is to mix vertical and horizontal scrolling to get diagonal scrolling. Horizontal scrolling is achieved by adding 1 to or subtracting 1 from the LMS operand. Vertical scrolling is achieved by adding the line length to or subtracting the line length from the LMS operand. Diagonal scrolling is achieved by executing both operations. There are four possible diagonal scroll directions. If, for example, the line length is 256 bytes and we wish to scroll down and to the right, we must add 256+(-1)=255 to each LMS operand in the display list. This is a two-byte add; the BASIC program example given above avoids the difficulties of two-byte address manipulations but most programs will not be so contrived. For truly fast two-dimensional scrolling assembly language will be necessary.

All sorts of weird arrangements are possible if we differentially manipulate the LMS bytes. Lines could scroll relative to each other, or hop over each other. Of course, some of this could be done with a conventional display but more data would have to be moved to do it. The real advantage of LMS scrolling is its speed. Instead of manipulating an entire screenful of data, many thousands of bytes in size, a program need only manipulate two or perhaps a few dozen bytes.

FINE SCROLLING

The second important scrolling facility of the ATARI 400/800™ is the fine scrolling capability. Fine scrolling is the capability of scrolling a pixel in steps smaller than the pixel size. Coarse scrolls proceed in steps equal to one pixel dimension; fine scrolls proceed in steps of one scan line vertically and one color clock horizontally. Fine scrolling can only carry so far; to get full fine scrolling over long distances on the screen we must couple fine scrolling with coarse scrolling. (Throughout this chapter the term pixel refers to an entire character, not to the smaller dots which make up a character.)

There are only two steps to implement fine scrolling. First, we set the fine scroll enable bits in the display list instruction bytes for the mode lines in which we want fine scrolling. (In most cases we want the entire screen to scroll so we set all the scroll enable bits in all the display list instruction bytes.) Bit D5 of the display list instruction is the vertical scroll enable bit; bit D4 of the display list instruction is the horizontal scroll enable bit. We then store the scrolling value we desire into the appropriate scrolling register. There are two scrolling registers, one for horizontal scrolling and one for vertical scrolling. The horizontal scroll register (HSCROL) is at $D404; the vertical scroll register (VSCROL) is at $D405. For horizontal scrolling, we store into HSCROL the number of color clocks by which we want the mode line scrolled. For vertical scrolling, we store into VSCROL the number of scan lines that we want the mode line scrolled. These scroll values will be applied to every line for which the respective fine scroll is enabled.

There are two complicating factors that we encounter when we use fine scrolling. Both arise from the fact that a partially scrolled display shows more information than a normal display. Consider for example what happens when we horizontally scroll a line by half a character to the left. There are 40 characters in the line. Half of the first character disappears off of the left edge of the screen. The 40th character scrolls to the left. What takes its place? Half of a new character should scroll in to take the place of the now scrolled 40th character. This character would be the 41st character. But there are only 40 characters in a normal line; what happens? If we have implemented coarse scrolling then the 41st character suddenly appears on the screen after the first character disappears off of the left edge. This sudden appearance is jerky and unsightly. The solution to this problem has already been built into the hardware. There are three display options for line widths: the narrow playfield (128 color clocks wide), the normal playfield (160 color clocks wide) and the wide playfield (192 color clocks wide). These options are set by setting appropriate bits in the DMACTL register. When using horizontal fine scrolling, ANTIC automatically retrieves more data from RAM than it displays. For example, if DMACTL is set for normal playfield, which in BASIC mode 0 has 40 bytes per line, then ANTIC will actually retrieve data at a rate appropriate to wide playfield---48 bytes per line. This will throw lines off horizontally if it is not taken into account. The problem does not manifest itself if the programmer has already organized screen RAM into long horizontal lines as in Figure 6.1.

The corresponding problem for vertical scrolling can be handled in either of two ways. The sloppy way is to ignore it. Then we will not get half-images at both ends of the display. Instead, the images at the bottom of the display will not scroll in properly; they will suddenly pop into view. The proper way takes very little work. To get proper fine scrolling into and out of the display region we must dedicate one mode line to act as a buffer. We do this by refraining from setting the vertical scroll bit in the display list instruction of the last mode line of the vertically scrolled zone. The window will now scroll without the unpleasant jerk. The screen image will be shortened by one mode line. An advantage of scrolling displays now becomes apparent. It is quite possible to create screen images that have more than 192 scan lines in the display. This could be disastrous with a static display, but with a scrolling display images which are above or below the displayed region can always be scrolled into view.

Fine scrolling will only scroll so far. The vertical limit for fine scrolling is 16 scan lines; the horizontal limit for fine scrolling is 16 color clocks. If we attempt to scroll beyond these limits, ANTIC simply ignores the higher bits of the scroll registers. To get full fine scrolling (in which the entire screen smoothly scrolls as far as we wish) we must couple fine scrolling with coarse scrolling. To do this we first fine scroll the image, keeping track of how far it has been scrolled. When the amount of fine scrolling equals the size of the pixel, we reset the fine scroll register to zero and execute a coarse scroll. Figure 6.2 illustrates the process:

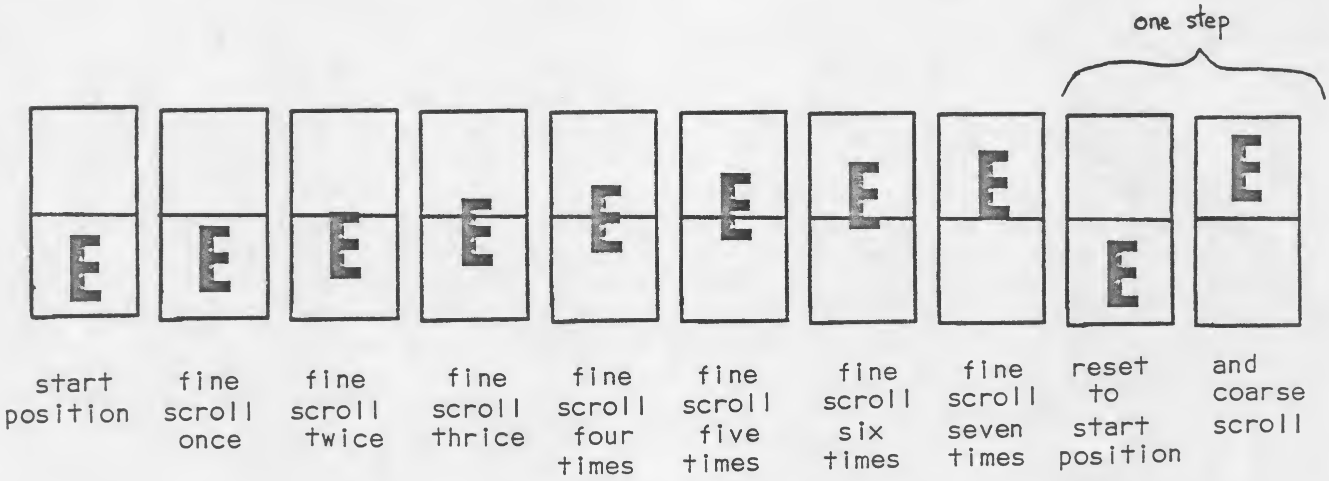
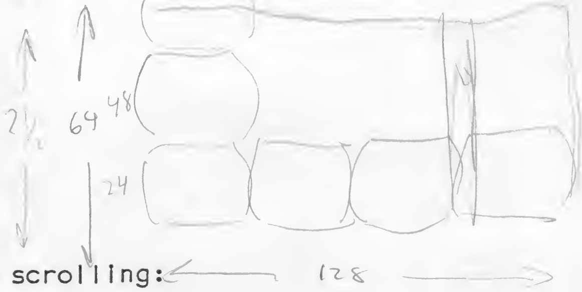


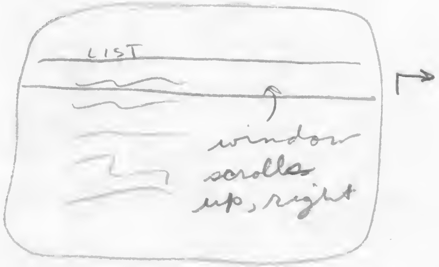
Figure 6.2
linking fine scroll to coarse scroll

SCROLLING



The following program illustrates simple fine scrolling:

```
1 HSCROL=54276
2 VSCROL=54277
10 GRAPHICS 0:LIST
20 DLIST=PEEK(560)+256*PEEK(561)
30 POKE DLIST+10,50:REM          enable both scrolls
40 POKE DLIST+11,50:REM          do it for two mode lines
50 FOR Y=0 TO 7
60 POKE VSCROL,Y:REM             vertical scroll
70 GOSUB 200:REM                 delay
80 NEXT Y
90 FOR X=0 TO 3
100 POKE HSCROL,X:REM            horizontal scroll
110 GOSUB 200:REM                delay
120 NEXT X
130 GOTO 40
200 FOR J=1 TO 200
210 NEXT J:RETURN
```



This program shows fine scrolling taking place at very slow speed. It demonstrates several problems that arise when using fine scrolling. First, the display lines below the scrolled window are shifted to the right. This is due to ANTIC's automatically retrieving 48 bytes per line instead of 40. The problem arises only in unrealistic demonstration programs such as this one; in real scrolling applications the arrangement of the screen data (as shown in Figure 6.1) precludes this problem. The second, more serious problem arises when the scroll registers are modified while ANTIC is in the middle of its display process. This confuses ANTIC and causes the screen to jerk. The solution is to change the scroll registers only during vertical blank periods. This can only be done with assembly language routines. Thus, fine scrolling normally requires the use of assembly language.

APPLICATIONS

The applications of full fine scrolling for graphics are numerous. The obvious application is for large maps which are created with character graphics. Using BASIC Graphics mode 2 I have created a very large map of Russia which contains about 10 screenfuls of image. The screen becomes a window to the map. The user can scroll about the entire map with a joystick. The system is very memory efficient; the entire map program plus data plus display list and character set definitions requires a total of about 4K of RAM.

There are many other applications of this technique. Any very large image that can be drawn with character graphics is amenable to this system. (Scrolling does not require character graphics. Map graphics are less desirable for scrolling applications because of their large memory requirements.) Large electronic schematics could be presented in this way.

The joystick could be used both to scroll around the schematic and to indicate particular components that the user wishes to address. Large blueprints or architectural diagrams could also be displayed with this technique. Any big image that need not be seen in its entirety can be presented with this system.

Large blocks of text are also usable here, although it might not be practical to read continous blocks of text by scrolling the image. This system is more suited to presenting blocks of independent text. One particularly exciting idea is to apply this system to menus. The program starts by presenting a welcome sign on the screen with signs indicating submenus pointing to other regions of the larger image. 'This way to addition' could point up while 'this way to subtraction' might point down. The user scrolls around the menu with the joystick, perusing his options. When he wishes to make a choice, he places a cursor on the option and presses the red button. Although this system could not be applied to all programs, it could be of great value to certain types of programs.

There are two 'blue sky' applications of fine scrolling which have not yet been fully explored. The first is selective fine scrolling, in which different mode lines of the display have different scroll bits enabled. Normally we would want the entire screen to scroll, but it is not necessary to do so. We could select one line for horizontal scrolling only, another line for vertical scrolling only, and so forth. The second blue sky feature is the prospect of using display list interrupts to change the HSCROL or VSCROL registers on the fly. Changing VSCROL on the fly is a tricky operation; it would probably confuse ANTIC and produce undesirable results. Changing HSCROL is also tricky but might be easier.

CHAPTER VII

ATARI BASIC OVERVIEW

This chapter discusses the ATARI BASIC. The four main topics are:

1. What is ATARI BASIC - a description of what is required to make BASIC run, and a discussion of its strengths and weaknesses.

2. How ATARI BASIC Works - a detailed analysis of how programs are tokenized and executed.

3. Improving Program Performance - a list of methods to increase the speed of a program and decrease its size.

4. Advanced Programming Techniques - a series of custom applications or "tricks" to meet various programming needs.

What Is ATARI BASIC

ATARI BASIC is like other BASICs in that it is an interpreted language. This means programs can be run when they are entered without intermediate stages of compilation and linking. The ATARI BASIC interpreter resides in an 8K ROM cartridge in the left slot of the computer. It encompasses addresses A000 through BFFF. BASIC stores the user's program in RAM and at least one 8K RAM board is required for this.

To use ATARI BASIC effectively one must know its strengths and weaknesses. With this information programs can be written that make good use of the assets and features of ATARI BASIC.

Strengths of ATARI BASIC:

1. It supports the operating system graphics - simple graphics calls can be made to display information on the screen.

2. It supports the hardware - such calls as SOUND, STICK and PADDLE are simple interfaces to the hardware of the computer.

3. Simple assembly interface - the USR function allows easy user access to assembly language routines.

4. ROM based interpreter - the BASIC interpreter is in ROM, which prevents accidental modification by the user program.

5. DOS support - specialized calls such as NOTE and POINT (DOS 2.0S) allow the user to randomly access a disk through the disk operating system.

6. Peripheral support - any peripheral recognized by the operating system can be accessed from a BASIC program.

Weaknesses of ATARI BASIC:

1. No support of integers - all numbers are stored as six byte BCD floating point numbers.

2. Slow math package - since all numbers are six bytes long, math operations become rather slow.

3. No string arrays - only one-dimensional strings can be created.

How ATARI BASIC Works

A brief overview of the workings of the BASIC interpreter is as follows:

1. BASIC gets a line of input from the user and converts it into a tokenized form.

2. It then puts this line into a token program.

3. This program is then executed.

The details of these operations are discussed in the following four sections.

A. THE TOKENIZING PROCESS

B. THE TOKEN FILE STRUCTURE

C. THE PROGRAM EXECUTION PROCESS

D. SYSTEM INTERACTION

A. THE TOKENIZING PROCESS

In simple terms, the tokenization of a line of code in BASIC looks like this:

1. BASIC gets a line of input
2. It then checks for legal syntax
3. During syntax checking it is tokenized
4. The tokenized line is moved into the token program
5. If the line is in immediate mode it is executed

To better understand the tokenizing process some terms must first be defined. These are:

Token - an 8 bit byte containing a particular interpretable code.

Statement - a complete "sentence" of tokens that causes BASIC to perform some meaningful task. In LIST form, statements are separated by colons.

Line - one or more statements preceeded either by a line number in the range of 0 to 32767, or an immediate mode line with no number.

Command - the first executable token of a statement that tells BASIC to interpret the tokens that follow in a particular way.

Variable - a token that is an indirect pointer to its actual value; thus the value can be changed without changing the token.

Constant - a six byte BCD value preceeded by a special token. This value remains unchanged throughout program execution.

Operator - any one of 46 tokens that in some way move or modify the values that follow them.

Function - a token that when executed returns a value to the program.

EOL - "End of Line", a character with the value 9B Hex.

BASIC begins the tokenizing process by getting a line of input. This input will be obtained from one of the handlers of the operating system. Normally it is from the screen editor, however with the ENTER command, any device can be specified. The call BASIC issues is a GET RECORD command, and the data returned is ATASCII information terminated by an EOL. This data is stored by CIO into the BASIC input line buffer from 580 to 5FF Hex.

After the record is returned the syntax checking and tokenizing processes begin. First BASIC looks for a line number. If one is found it is converted into a two byte integer. If no line number is present it is assumed to be in immediate mode and the line number 8000 Hex is assigned to it. These will be the first two tokens of the tokenized line. This line is built in the token output buffer that is 256 byte long and resides at the end of the reserved operating system RAM.

The next token is a dummy byte reserved for the byte count (or offset) from the start of this line to the start of the next line. Following that is another dummy byte for the count of the start of this line to the start of the next statement. These values will be set when tokenization is complete for the line and the statement respectively. The use of these values is discussed in the program execution process section.

BASIC now looks for the command of the first statement of the input line. A check is made to determine if this is a valid command by scanning a list of legal commands in ROM. If a match is found then the next byte in the token line becomes the number of the entry in the ROM list that matched. If no match is found a syntax error token is assigned to that byte and BASIC stops tokenizing, copies the rest of the input buffer in ATASCII format to the token output buffer, and prints the error line.

Assuming a good line, following the command can be one of seven items: a variable, a constant, an operator, a function, a double quote, another statement, or an EOL. BASIC tests if the next input character is numeric. If not then it compares that character and those following against the entries of the variable name table. If this is the first line of code entered in the program then no match will be found. The characters are then compared against the function and operator tables. If no match is found there then BASIC assumes that this is a new variable name. Since this is the first variable it will be assigned the first entry in the variable name table. The characters are copied out of the input buffer and stored into the name table with the most significant bit (MSB) set on the last byte of the name. Eight bytes are then reserved in the variable value table for this entry. (See the variable value table discussion in the token file structure section.)

The token that ends up in the tokenized line is the variable number minus one, with the MSB set. Thus the token of the first variable entered would be 80 Hex, the second would be 81, and so on up to FF for a total of 128 unique variable numbers.

If a function is found, then its entry number in the operator function table is assigned to the token. Functions require certain sequences of parameters; these are contained in syntax tables, and if they are not matched then a syntax error will result.

If an operator is found, then a token is given its table entry number. Operators can follow each other in a rather complex fashion (such as multiple parentheses) so the syntax checking of them is a bit complicated.

In the case of the double quotes, BASIC assumes that a character string is following and assigns a 0F Hex to the output token and reserves a dummy byte for the string length. The characters are moved from the input buffer into the output buffer until the second set of quotes is found. The length byte is then set to the character count.

If the next characters in the input buffer are numeric, BASIC will convert them into a six byte BCD constant. A 0E Hex token will be put in the output buffer, followed by the six byte constant.

When a colon is encountered, a 14 Hex token is inserted in the output buffer and the offset from the start of the line is stored in the dummy byte that was reserved for the count to the start of the next statement. At this point another dummy byte is reserved and the process goes back to get a command.

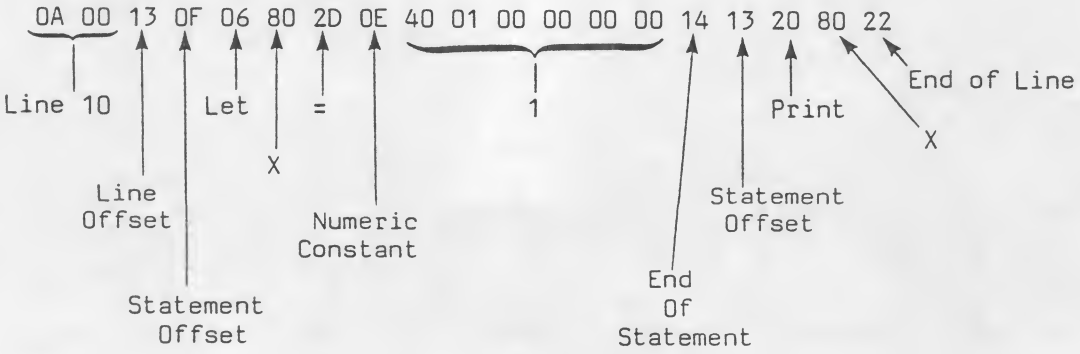
When the EOL is found, a 16 Hex token is stored and the offset from the start of the line is put in the dummy byte for the line offset. At this point tokenization is complete and BASIC will move the token line into the token program. First it searches the program for that line number. If it is found it replaces the old line with the new one. If it is not found then the new line is inserted in the correct numerical sequence. In both cases, the data following the line will either be moved up or down in memory to allow for an expanding and contracting program size.

BASIC now checks if the tokenized line is an immediate mode line. If so, that line is executed according to the methods described in the interpretive process, if not then BASIC goes back to get another line of input.

If at any time during the tokenizing process the length of the token line exceeds 256 bytes, an error 14 message (line too long) is sent to the screen and BASIC will go back to get the next line of input.

An example line of input and its token form looks like this (all token values are hexadecimal):

```
10 LET X=1 : PRINT X
```



COMMANDS			OPERATORS			FUNCTIONS		
HEX	DEC		HEX	DEC		HEX	DEC	
00	0	REM	0E	14	[NUM CONST]	3D	61	STR$
01	1	DATA	0F	15	[STR CONST]	3E	62	CHR$
02	2	INPUT	10	16	"	3F	63	USR
03	3	COLOR	11	17	[NOT USED]	40	64	ASC
04	4	LIST	12	18	,	41	65	VAL
05	5	ENTER	13	19	$	42	66	LEN
06	6	LET	14	20	: [STMT END]	43	67	ADR
07	7	IF	15	21	;	44	68	ATN
08	8	FOR	16	22	[LINE END]	45	69	COS
09	9	NEXT	17	23	GOTO	46	70	PEEK
0A	10	GOTO	18	24	GOSUB	47	71	SIN
0B	11	GO TO	19	25	TO	48	72	RND
0C	12	GOSUB	1A	26	STEP	49	73	FRE
0D	13	TRAP	1B	27	THEN	4A	74	EXP
0E	14	BYE	1C	28	#	4B	75	LOG
0F	15	CONT	1D	29	<= [NUMERICS]	4C	76	CLOG
10	16	COM	1E	30	<>	4D	77	SQR
11	17	CLOSE	1F	31	>=	4E	78	SGN
12	18	CLR	20	32	<	4F	79	ABS
13	19	DEG	21	33	>	50	80	INT
14	20	DIM	22	34	=	51	81	PADDLE
15	21	END	23	35	°	52	82	STICK
16	22	NEW	24	36	*	53	83	PTRIG
17	23	OPEN	25	37	+	54	84	STRIG
18	24	LOAD	26	38	-			
19	25	SAVE	27	39	/			
1A	26	STATUS	28	40	NOT			
1B	27	NOTE	29	41	OR			
1C	28	POINT	2A	42	AND			
1D	29	XIO	2B	43	(			
1E	30	ON	2C	44	)			
1F	31	POKE	2D	45	= [ARITHM ASSIGN]			
20	32	PRINT	2E	46	= [STRING ASSIGN]			
21	33	RAD	2F	47	<= [STRINGS]			
22	34	READ	30	48	<>			
23	35	RESTORE	31	49	>=			
24	36	RETURN	32	50	<			
25	37	RUN	33	51	>			
26	38	STOP	34	52	=			
27	39	POP	35	53	+ [UNARY]			
28	40	?	36	54	-			
29	41	GET	37	55	([STRING LEFT PAREN]			
2A	42	PUT	38	56	([ARRAY LEFT PAREN]			
2B	43	GRAPHICS	39	57	([DIM ARRAY LEFT PAREN]			
2C	44	PLOT	3A	58	([FUN LEFT PAREN]			
2D	45	POSITION	3B	59	([DIM STR LEFT PAREN]			
2E	46	DOS	3C	60	, [ARRAY COMMA]			
2F	47	DRAWTO						
30	48	SETCOLOR						
31	49	LOCATE						
32	50	SOUND						
33	51	LPRINT						
34	52	CSAVE						
35	53	CLOAD						
36	54	[IMPLIED LET]						
37	55	ERROR- [SYNTAX]						

B. THE TOKEN FILE STRUCTURE

The token file contains two major segments: 1) a group of zero page pointers that point into the token file, and 2) the actual token file itself. The zero page pointers are 2-byte values that point to various sections of the token file. There are nine 2-byte pointers and they are in locations 80 to 91 Hex. Following is a list of the pointers and the sections of the token file they reference.

POINTER (Hex)	TOKEN FILE SECTION (Contiguous Blocks)
LOMEM 80,81	Token output buffer - this is the buffer BASIC uses to tokenize one line of code. It is 256 bytes long. This buffer resides at the end of the operating system's allocated RAM.
VNTP 82,83	Variable Name Table - a list of all the variable names that have been entered in the program. They are stored as ATASCII characters, each new name stored in the order it was entered. Three types of name entries exist: 1. Scalar variables - MSB set on last character in name. 2. String variables - last character is a "$" with the MSB set. 3. Array variables - last character is a "(" with the MSB set.
VNTD 84,85	Variable Name Table dummy end - BASIC uses this pointer to indicate the end of the name table. This normally points to a dummy zero byte when there are less than 128 variables. When 128 variables are present, this points to the last byte of the last variable name.
VVTP 86,87	Variable Value Table - this table contains current information on each variable. For each variable in the name table, 8 bytes are reserved in the value table. The information for each variable type is:

BYTE NUMBER	1	2	3 4	5 6	7 8
SCALAR	00	Var#	Six byte BCD constant		
ARRAY (DIMed) (unDIMed)	41 40	Var#	Offset from STARP(8C,8D)	first DIM + 1	second DIM + 1
STRING (DIMed) (unDIMed)	81 80	Var#	Offset from STARP(8C,8D)	Length	DIM

A scalar variable contains a numeric value. An example is X=1. The scalar is X and its value is 1, stored in six byte BCD format. An array is composed of numeric elements stored in the string/array area and has one entry in the value table. A string, composed of character elements in the string/array area, also has one entry in the table.

POINTER (Hex) TOKEN FILE SECTION (Contiguous Blocks)

The first byte of each value entry indicates the type of variable: 00 for a scalar, 40 for an array, and 80 for a string. If the array or string has been dimensioned, then the LSB is set on the first byte.

The second byte contains the variable number. The first variable entry is number zero and if 128 variables were present the last would be 7F.

In the case of the scalar variable the third through eighth byte contain the six byte BCD number that has currently been assigned to it.

For arrays and strings, the third and fourth bytes contain an offset from the start of the string/array area (described below) to the beginning of the data.

The fifth and sixth bytes of an array contain its first dimension. The quantity is a two byte integer and its value is 1 greater than the user entered. The seventh and eighth bytes are the second dimension, also a value of 1 greater.

The fifth and sixth bytes of a string are a two byte integer that contains its current length. The seventh and eighth bytes are its dimensions.

STMTAB 88,89 Statement Table - this block of data includes all the lines of code that have been entered by the user and tokenized by BASIC, and it also includes the immediate mode line. The format of these lines is described in the tokenized line example of the section on the tokenizing process.

STMCUR 8A,8B Current Statement - this pointer is used by BASIC to reference particular tokens within a line of the statement table. When BASIC is waiting for input, this pointer is set to the beginning of the immediate mode line.

STARP 8C,8D String/Array area - this block contains all the string and array data. String characters are stored as one byte ATASCII entries, so a string of 20 characters will require 20 bytes. Arrays are stored with 6 byte BCD numbers for each element. A 10 element array will require 60 bytes.

This area is allocated and subsequently enlarged by each dimension statement encountered, the amount being equal to the size of a string dimension or six times the size of an array dimension.

POINTER (Hex) TOKEN FILE SECTION (Contiguous Blocks)

RUNSTK 8E,8F

Run Time Stack - this software stack contains GOSUB and FOR/NEXT entries. The GOSUB entry consists of four bytes. The first is a 0 byte indicating GOSUB, followed by the two byte integer line number on which the call occured, followed by the offset into that line so the RETURN can come back and execute the next statement.

The FOR/NEXT entry contains 16 bytes. The first is the limit the counter variable can reach. The second byte is the step or counter increment. Each of these quantities is in 6 byte BCD format. The thirteenth byte is the counter variable number with the MSB set, the fourteenth and fifteenth bytes are the line number, and the sixteenth is the line offset to the FOR statement.

MEMTOP 90,91

Top of Application RAM - this is the end of the user program. Program expansion can occur from this point to the end of free RAM, which is defined by the start of the display list. The FRE function returns the amount of free RAM by subtracting MEMTOP from HIMEM (2E5,2E6). Note that the BASIC MEMTOP is not the same as the OS variable called MEMTOP.

C. THE PROGRAM EXECUTION PROCESS

The process of executing a line of code involves reading the tokens that were created during the tokenization process. Each token has a particular meaning that causes BASIC to execute a specific series of operations. The method of doing this requires that BASIC get one token at a time from the token program and then process it. The token is an index into a jump table of routines, so a PRINT token will point indirectly to a PRINT processing routine. When that processing is complete, BASIC returns to get the next token. The pointer that is used to fetch each token is called STMCUR and is at 8A and 8B.

The first line of code that is executed in a program is the immediate mode line. This is usually a RUN or GOTO. In the case of the RUN, BASIC gets the first line of tokens from the statement table (tokenized program) and processes it. If all the code is in-line, then BASIC will merely execute consecutive lines.

If a GOTO is encountered, then the line to go to must be found. The statement table contains a partially linked list of line numbers and statements, the lowest line number first, followed by increasing line numbers up to the largest. If a line somewhere in the middle of the table is needed, the process is as follows:

> The address of the first line is found in the STMTAB pointer at 88 and 89. This is stored in a temporary pointer. The first two bytes of the first line are its line number. This number is compared against the requested line number. If the first number is less, then BASIC gets the next line by adding the third byte of the first line to the temporary pointer. The temporary pointer will be pointing to the second line. Again the first two bytes of this new line are compared to the requested line, and if they are less, the third byte is added to the pointer. If a line number does match, then the contents of the temporary pointer are moved into STMCUR and BASIC will fetch the next token from the new line. Should the requested line number not be found, then an Error 12 will be generated.

The GOSUB involves more processing than the GOTO. The line finding routine is the same, but before BASIC goes to that line it sets up an entry in the Run Time Stack. It allocates four bytes at the end of the stack and stores a 0 in the first byte to indicate a GOSUB stack entry. It then stores the line number it was on when the call was made into the next two bytes of the stack. The final byte contains the offset in bytes from the start of that line to where the GOSUB token was found. BASIC then executes the line it looked up. When the RETURN is found, the entry on the stack is pulled off, and BASIC returns to the calling line.

The FOR command causes BASIC to allocate 16 bytes on the Run Time Stack. The first six bytes are the limit the variable can reach in six byte BCD format. The second six bytes are the step, in the same format. Following these, BASIC stores the variable number (MSB set) of the counting variable. It then stores the present line number (2 bytes) and the offset into the line. The rest of the line is then executed.

When BASIC finds the NEXT command, it looks at the last entry on the stack. It makes sure the variable referenced by the NEXT is the same as the one on the stack and checks if the counter has reached or exceeded the limit. If not then BASIC returns to the line with the FOR statement and continues execution. If the limit was reached then the FOR entry is pulled off the stack and execution continues from that point.

When an expression is evaluated, the operators are put onto an operator stack and are pulled off one at a time and evaluated. The order in which the operators are put onto the stack can either be implied, in which case BASIC looks up the operator's precedence from a ROM table, or the order can be explicitly stated by the placement of parentheses.

If at any time the BREAK key is hit, the operating system sets a flag to indicate this occurrence. BASIC checks this flag after each token is processed. If it finds it has been set, it stores the line number at which this occured, prints out a "STOPPED AT LINE number" message, clears the BREAK flag and waits for user input. At this point the user could type CONT and program execution would continue at the next line.

D. SYSTEM INTERACTION

BASIC communicates with the operating system primarily through the use of I/O calls to the Central I/O Utility (CIO). Following is a list of user BASIC calls and the corresponding operating system IOCBs.

BASIC	OS

OPEN #1,12,0,"E:"	IOCB=1
	Command=3 (OPEN)
	Aux1=12 (Input/Output)
	Aux2=0
	Buffer Address=ADR("E:")

GET #1,X	IOCB=1
	Command=7 (Get Characters)
	Buffer Length=0
	Character returned in accumulator

PUT #1,X	IOCB=1
	Command=11 (Put Characters)
	Buffer Length=0
	Character output through accumulator

INPUT #1,A$	IOCB=1
	Command=5 (Get Record)
	Buffer Length=Length of A$ (not over 120)
	Buffer Address=Input Line Buffer

PRINT #1,A$	IOCB=1 BASIC uses a special put byte vector in the IOCB to talk directly to the handler.

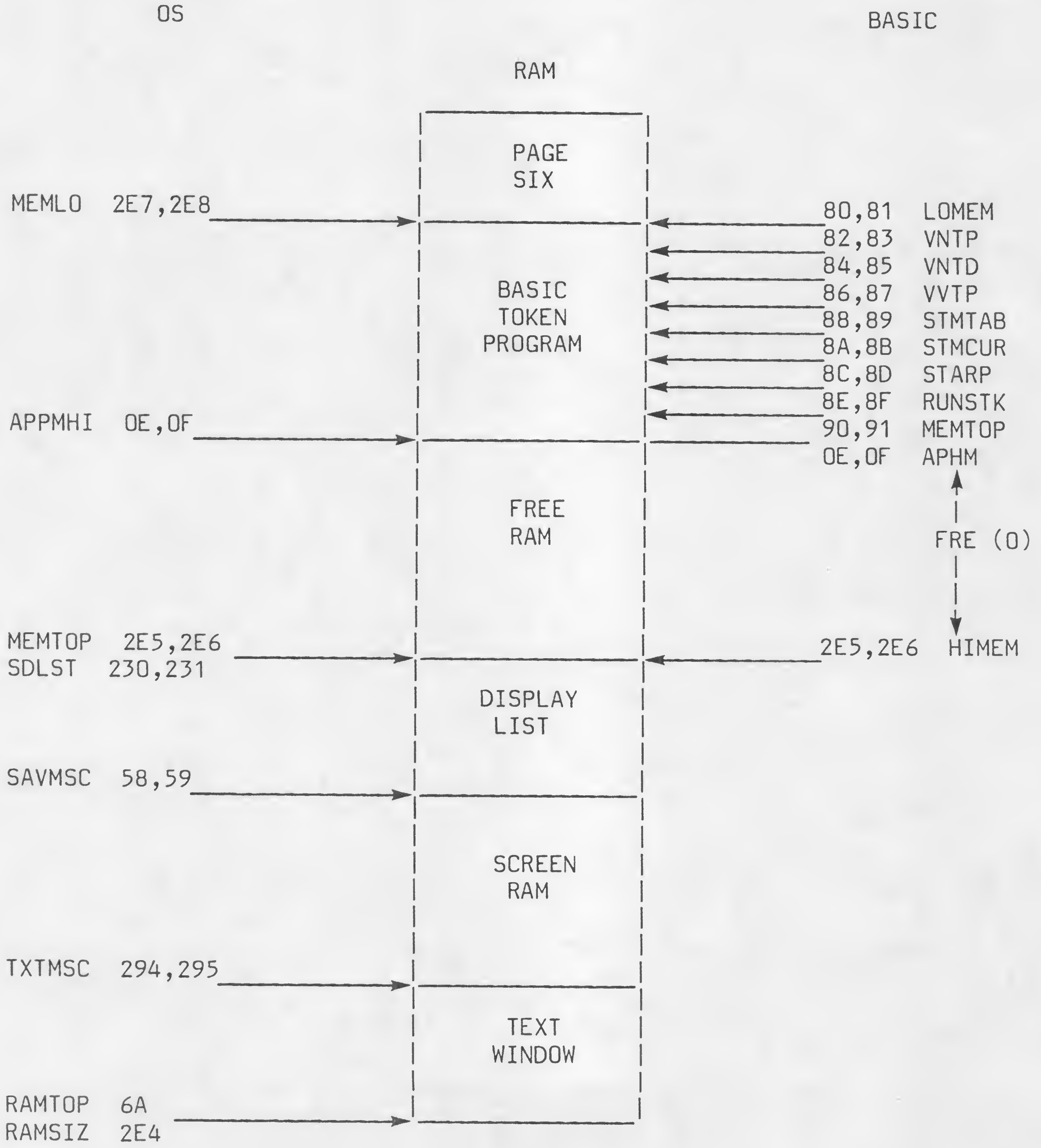
XIO 18,#6,12,0,"S:"	IOCB=6
	Command=18 (Special - Fill)
	Aux1=12
	Aux2=0

SAVE/LOAD: When a BASIC token program is SAVEd to a device, two blocks of information are written. The first block consists of seven of the nine zero page pointers that BASIC uses to maintain the token file. These are LOMEM(80,81) through STARP (8C,8D). There is one change made to these pointers when they are written out: The value of LOMEM is subtracted from each of the two-byte pointers, and these new values are written to the device. Thus the first two bytes written will be 0,0.

The second block of information written consists of the following token file sections: 1) The variable name table, 2) the variable value table, 3) the token program, and 4) the immediate mode line.

When this program is LOADed into memory, BASIC looks at the OS variable MEMLO (2E7,2E8) and adds its value to each of the two-byte zero page pointers as they are read from the device. These pointers are placed back on page zero and then the values of RUNSTK (8E,8F) and MEMTOP (90,91) are set to the value in STARP.

Next, 256 bytes are reserved in memory above the value of MEMLO to allocate space for the token output buffer. Then the token file information, consisting of the variable name table through the immediate mode line, is read in. This data is placed in memory immediately following the token output buffer.



OS AND BASIC POINTERS (NO DOS PRESENT)

Improving Program Performance

Program performance can be improved in two ways. First the execution time can be decreased (it will run faster) and second, the amount of space required can be decreased, allowing it to use less RAM. To attain these two goals, the following lists can be used as guidelines. The methods of improvement in each list are primarily arranged in order of decreasing effectiveness. Therefore the method at the top of a list will have more impact than one on the bottom.

Speeding Up A BASIC Program:

1. Recode - since BASIC is not a structured language, the code written in it tends to be inefficient. After many revisions it becomes even worse. Thus, spending the time to restructure the code is worthwhile.

2. Check algorithm logic - make sure that the code to execute a process is as efficient as possible.

3. Put frequently called subroutines and FOR/NEXT loops at the start of the program - BASIC starts at the beginning of a program to look for a line number, so any line references near the end will take longer to reach.

4. For frequently called operations within a loop, use in-line code rather than subroutines - the program speed can be improved here since BASIC spends time adding and removing entries from the run time stack.

5. Make the most frequently changing loop of a nested set the deepest - in this way the run time stack will be altered the fewest number of times.

6. Simplify floating point calculations within the loop - if a result is obtained by multiplying a constant by a counter, time could be saved by changing the operation to an add of a constant.

7. Set up loops as multiple statements on one line - in this way the BASIC interpreter will not have to get the next line to continue the loop.

8. Disable the screen display - if visual information is not important for a period of time, up to a 30% time savings can be made with a POKE 559,0.

9. Use a faster graphics mode or a short display list - if a full screen display is not necessary then up to 25% time savings can be made.

10. Use assembly code - time savings can be made by encoding loops in assembler and using the USR function.

Saving Space In A BASIC Program:

1) Recode - as mentioned previously, restructuring the program will make it more efficient. It will also save space.

2) Remove remarks - remarks are stored as ATASCII data and merely take up space in the running program.

3) Replace a constant used three times or more with a variable - BASIC allocates seven bytes for a constant but only one for a variable reference, so six bytes can be saved each time a constant is replaced with a variable assigned to that constant's value.

4) Initialize variables with a read statement - a data statement is stored in ATASCII code, one byte per character, whereas an assignment statement requires seven bytes for one constant.

5) Try to convert numbers used once and twice to operations of predefined variables - an example is to define Z1 to equal 1, Z2 to equal 2, and if the number 3 is required, replace it with the expression Z1 + Z2.

6) Set frequently used line numbers (in GOSUB and GOTO) to predefined variables - if the line 100 is referenced 50 times, approximately 300 bytes can be saved by equating Z100 to 100 and referencing Z100.

7) Keep the number of variables to a minimum - each new variable entry requires 8 more bytes in the variable value table plus a few bytes for its name.

8) Clean up the value and name tables - variable entries are not deleted from the value and name tables even after all references to them are removed from the program. To delete the entries LIST the program to disk or cassette, type NEW, then ENTER the program.

9) Keep variable names as short as possible - each variable name is stored in the name table as ATASCII information. The shorter the names, the shorter the table.

10) Replace text used repeatedly with strings - on screens with a lot of text, space can be saved by assigning a string to a commonly used set of characters.

11) Initialize strings with assignment statements - an assignment of a string with data in quotes requires less space than a READ statement and a CHR$ function.

12) Concatenate lines into multiple statements - three bytes can be saved each time two lines are converted into two statements on one line.

13) Replace once used subroutines with in-line code - the GOSUB and RETURN statements waste bytes if used only once.

14) Replace numeric arrays with strings if the data values do not exceed 255 - numeric array entries require six bytes each, whereas string elements only need one.

15) Replace set color statements with POKE commands - this will save 8 bytes.

16) Use cursor control characters rather than POSITION statements - the POSITION statement requires 15 bytes for the X,Y parameters whereas the cursor editing characters are one byte each.

17) Delete lines of code via program control - see the advanced programming techniques section.

18) Modify the string/array pointer to load predefined data - see the advanced programming techniques section.

19) Small assembly routines can be stored in remark statements - remarks are stored as unchanged ATASCII data.

20) Chain programs - an example would be an initialization routine that is run first, and which then loads and runs the main program.

Advanced Programming Techniques

When the fundamentals of ATARI BASIC are understood some interesting applications can be written. These can be strictly BASIC operations, or they can also involve features of the operating system.

Example 1 - String Initialization - This program will set all the bytes of a string of any length to the same value. BASIC copies the first byte of the source string into the first byte of the destination string, then the second, third, and so on. By making the destination string the second byte of the source, the same character can be stored into the entire string.

Example 2 - Delete Lines Of Code - By using a feature of the operating system, a program can delete or modify lines of code within itself. The screen editor can be set to accept data from the screen without user input. Thus by first setting up the screen, positioning the cursor to the top, and then stopping the program, BASIC will be getting the commands that have been printed on the screen.

Example 3 - Saving The String/Array Area - If an array or string is always initialized to the same size and data, then an appreciable amount of program space can be saved by storing the information during the SAVE and deleting the initialization code for the next run.

Example 4 - Save BCD Numbers To Disk - Whenever numeric data is written to a device it is sent as ATASCII information. This means the number 10 is written as an ATASCII 1 followed by a 0. This makes a mess out of fixed length records. One way to correct this is to store the six byte BCD number to disk directly by equating it to a string, and then writing that string. It can be retrieved in the same way.

Example 5 - Player/Missile Graphics With Strings - A fast way to move player/missile graphics data is shown in this example. A dimensioned string has its string/array area offset value changed to point to the P/M graphics area. Writing to this string with an assignment statement will now write data into the P/M area at assembly language rates.

```
10 REM STRING INITIALIZATION
20 DIM A$(1000)
30 A$(1)="A":A$(1000)="A"
40 A$(2)=A$
```

```
10 REM DELETE LINE EXAMPLE
20 GRAPHICS 0:POSITION 2,4
30 ? 70:? 80:? 90:? "CONT"
40 POSITION 2,0
50 POKE 842,13:STOP
60 POKE 842,12
70 REM THESE LINES
80 REM WILL BE
90 REM DELETED
```

```
10 REM STRING/ARRAY SAVE
15 REM GOTO 20 FOR FIRST RUN
17 REM DELETE LINE 20 FOR SECOND RUN
18 GOTO 100
20 DIM A$(10):A$="WWWWWWWWWW"
30 STARP=PEEK(140)+PEEK(141)*256
40 STARP=STARP+10
50 HI=INT(STARP/256):LO=STARP-HI*256
60 POKE 140,LO:POKE 141,HI
70 SAVE "D:STRING":STOP
100 STARP=PEEK(140)+PEEK(141)*256
110 STARP=STARP-10
120 HI=INT(STARP/256):LO=STARP-HI*256
130 POKE 140,LO:POKE 142,LO:POKE 144,LO
140 POKE 141,HI:POKE 143,HI:POKE 145,HI
150 DIM A$(10)
160 A$(10,10)="W"
170 STOP
```

```
10 REM SAVE AND RETRIEVE BCD NUMBERS ON DISK
15 DIM A(0),B$(6)
20 B$(6,6)=CHR$(32)
30 VTAB=PEEK(134)+PEEK(135)*256
40 POKE VTAB+10,0
50 OPEN #1,8,0,"D:TEST"
60 FOR C=1 TO 15:A(0)=C:? #1;A$:NEXT C
70 CLOSE #1
80 OPEN #1,4,0,"D:TEST"
90 FOR C=1 TO 15:INPUT #1,A$:? A(0):NEXT C
100 CLOSE #1:END
```

```
100 REM PLAYER/MISSILE EXAMPLE
110 DIM A$(512),B$(20)
120 X=X+1:READ A:IF A<>-1 THEN B$(X,X)=CHR$(A):GOTO 120
130 DATA 0,255,129,129,129,129,129,129,129,129,255,0,-1
2000 POKE 559,62:POKE 704,88
2020 I=PEEK(106)-16:POKE 54279,I
2030 POKE 53277,3:POKE 710,224
2040 VTAB=PEEK(134)+PEEK(135)*256
2050 ATAB=PEEK(140)+PEEK(141)*256
2060 OFFS=I*256+1024-ATAB
2070 HI=INT(OFFS/256):LO=OFFS-HI*256
2090 POKE VTAB+2,LO:POKE VTAB+3,HI
3000 Y=60:Z=100:V=1:H=1
4000 A$(Y,Y+11)=B$:POKE 53248,Z
4010 Y=Y+V:Z=Z+H
4020 IF Y>213 OR Y<33 THEN V=-V
4030 IF Z>206 OR Z<49 THEN H=-H
4420 GOTO 4000
```

CHAPTER 8
OPERATING SYSTEM

INTRODUCTION TO THE OPERATING SYSTEM

This section is a simple introduction to the Operating System (O.S.) of the ATARI 400/800. This section also contains a brief description of the elements of the O.S. The following sections will describe these elements in detail:

Input/Output Subsystem
ROM-Based Character Set
System Routine's Vectors
Interrupt Handlers
Monitor
System Database
System and Event Timers
Floating Point Package

The O.S. allows the application programmer access to the full capabilities of the computer's hardware. The ATARI Personal Computer System's hardware is capable of providing you with some excellent I/O functions via the I/O subsystem. The I/O subsystem is a set of system routines that interface with the I/O hardware. The rest of the O.S. supports the I/O subsystem and provides you with additional features you can use for applications.

The ROM-based character set is used by the display handlers to write characters on the television screen. If you wish, you can create your own character set and tell the O.S. to use it instead.

The system vectors provide the glue that holds the O.S. together. The O.S. uses the vectors to move from one execution environment (BASIC, DUP, Star Raiders™) to another. You can call any system routine by jumping to its vector. The vectors are most frequently used to call I/O system routines, set timers and transfer control to different execution environments.

The system routines may be vectored in one of two ways. The ROM vectors are locations that contain JMP instructions to system routines and cannot be altered. The RAM vectors are locations that contain alterable addresses of system routines. The locations of both types of vectors are guaranteed to remain the same in future releases of the O.S.

The computer generates interrupts for several different reasons. Some of the most common interrupts are keyboard, <BREAK>, serial bus, and vertical blank.

The O.S. monitor is a system routine to initialize the computer on power-up and system reset. The monitor initializes the I/O subsystem, sets up the vectors and selects the execution environment after initialization is complete.

The O.S. is supported by a large database consisting of various system flags, I/O buffers, and screen/graphics registers. Most of the database is dedicated to the I/O subsystem. The database also contains locations used by the other parts of the O.S. Application programmers can take advantage of the database to add features to their programs.

The O.S. has two types of timers, system timers and hardware timers. System timers are used by applications programs as general purpose software timers. Hardware timers are used to time 'real time' events such as the television scan lines in a screen display.

The floating point package is a set of mathematical routines available to the user. The routines use binary coded decimal (BCD) arithmetic. Routines are provided to do the standard arithmetic functions (+,-,*,/) as well as conversion from ATASCII to BCD and BCD to ATASCII. You can find a description of the floating point package as well as how to use it in Section 8 of the O.S. User Manual. Appendix 5 of this book contains an example using the package.

[THIS PAGE INTENTIONALLY LEFT BLANK]

C

OPERATING SYSTEM
I/O SUBSYSTEM STRUCTURE

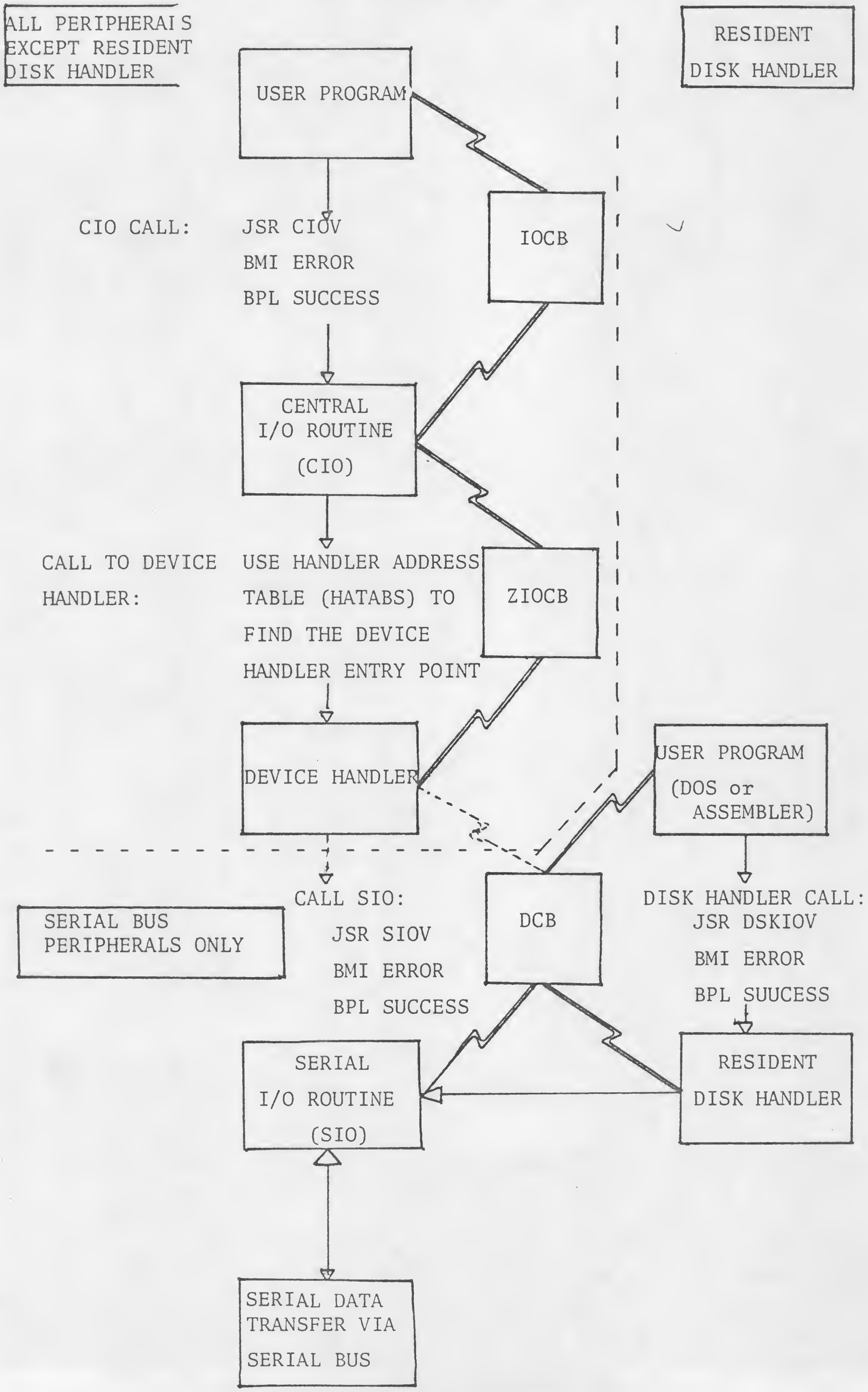


FIGURE 8.1 I/O SUBSYSTEM

SECTION 8.2 - - I/O SUBSYSTEM

INTRODUCTION

The I/O subsystem provides a convenient method of accessing the I/O hardware registers used by ANTIC, POKEY, CTIA and PIA. These special purpose chips control the I/O devices such as the keyboard, printer and disk. You simply pass control data to the I/O subsystem and it will perform the requested I/O function for that device.

The I/O subsystem has two types of elements: I/O system routines and I/O system control blocks. The I/O system routines are the central I/O routine (CIO), the device handlers (i.e. E:, P:, K:) and the serial I/O routine (SIO). The system I/O control blocks contain control data that is routed to the I/O subsystem. The user interface appears the same for all devices (e.g. the commands to output a line to the printer (P:) or to the display editor (E:) are very similar).

You need to understand the structure of the I/O subsystem to get the most out of it. Figure 8.1 shows the relationship of the I/O system routines and the I/O system control blocks.

I/O SYSTEM CONTROL BLOCKS

There are three types of control blocks:

Input/Output Control Block (IOCB)
Zero-Page I/O Control Block (ZIOCB)
Device Control Block (DCB)

The I/O system control blocks are used to communicate information about the I/O function to be executed. The control blocks provide the I/O system routines with control information to perform the I/O function. The O.S. Manual has information as to the detailed structure of the three types of control blocks in Section 5.

DCB CHART

FUNCTION	NAME	LOCATION	DISK 810/815						PRINTER 820
			READ SECTOR		WRITE SECTOR		PUT	FORMAT	WRITE
			810	815	810	815			
Serial Bus I.D.	DDEVIC	[$0300]	$30	$30	$30	$30	$30	$30	$40
Device Number	DUNIT	[$0301]	1-4	1-8	1-4	1-8	1-4	1-4	1
Command Byte	DCOMND	[$0302]	$52	$52	$57	$57	$50	$21	$57
Status	DSTATS	[$0303]	$40	$40	$80	$80	$80	$40	$80
Buffer Address	DBUFLO	[$0304]	U	U	U	U	U	U	U
	DVBUFHI	[$0305]	U	U	U	U	U	U	U
Device Timeout	DTIMLO	[$0306]	$30	$30	$30	$30	$31	$130	5
Buffer Length	DBYTLO	[$0308]	$80	00	$80	00	$80/00	-	$40
	DBYTHI	[$0309]	$00	01	$00	01	$00/01	-	$00
	DAUX1	[$030A]	2*	2*	2*	2*	- 2*	-	1*
	DAUX2	[$030B]	2*	2*	2*	2*	- 2*	-	1*

FIGURE 8.3

1* = This byte determines printer mode (see 820 manual.
2* = DAUX1 + DAUX2 specify sector for READ, WRITE (PUT), or WRITE verify.
U = Indicates user-set address
- = indicates ignored.

IOCB CHART

CALL	ICHID	ICDNO	ICCOM	ICSTA	ICBAL	ICBAH	ICPTL	ICPTH	ICBLL	ICBLH	ICAX1	ICAX2
OPEN FILE-READ	X	X	3	note 1	$80	06	X	X	X	X	4	0
OPEN FILE-WRITE	X	X	3	"	$80	06	X	X	X	X	8	note 2
GET BYTES	X	X	7	"	00	06	X	X	$80	00	X	X
PUT BYTES	X	X	$B	"	00	06	X	X	$80	00	X	X
GET RECORD	X	X	5	"	00	06	X	X	$80	00	X	X
PUT RECORD	X	X	9	"	00	06	X	X	$80	00	X	X
CLOSE FILE	X	X	$C	"	X	X	X	X	X	X	X	X
STATUS	X	X	$D	"	X	X	X	X	X	X	X	X

FIGURE 8.2

NOTE 1 = The status of the I/O command is stored here and in the Y REG. on return from CIO.
NOTE 2 = The Auxilary bytes of the IOCB's are used by some handlers to indicate special modes.
X = Indicates ignore but do not change the current value.
GENERAL NOTE: The above IOCB definitions assume:

```
*=$600
IOBUFF    .RES     80                   USER    I/O BUFFER
FILE      .BYTE    'D:MYPROG.BAS'       USER    FILENAME
```

The eight IOCBs in the O.S. are used for communication between user programs and CIO. Figure 8.2 shows the content of an IOCB for some common I/O functions. The IOCB's are:

Name	Location, Length
IOCB0	[$340,16]
IOCB1	[$350,16]
IOCB2	[$360,16]
IOCB3	[$370,16]
IOCB4	[$380,16]
IOCB4	[$390,16]
IOCB5	[$3A0,16]
IOCB6	[$3B0,16]
IOCB7	[$3C0,16]

The second type of control block, the ZIOCB [$0020,16], is used to communicate I/O control data between CIO and the device handlers. When called, CIO uses the value contained in the X register as an index that points to the starting address of the IOCB (one of 8) to be used. CIO then moves the control data from the selected IOCB to the ZIOCB for use by the appropriate device handler. The ZIOCB is of little interest unless you are writing a new device handler or are replacing a current one. See section 9 of the O.S. User Manual for more information on the ZIOCB.

Device Control Block

The device handlers load control information into the DCB [$0300,16]. SIO will use the DCB information and return the status information in the DCB for subsequent use by the device handler. Only device handlers that use the serial bus use the DCB and SIO. Section 9 of the O.S. User Manual contains a detailed description of the DCB. Figure 8.3 illustrates some common I/O functions and the contents of their associated DCBs.

The resident disk handler does not conform to the regular user-CIO-handler-SIO calling sequence. Instead, you use the DCB to communicate directly with the resident disk handler. Chapter 9 of this book contains more information on the resident disk handler.

CENTRAL I/O SYSTEM ROUTINE - CIO

CIO's main function is to route I/O control data to the correct device handler and then pass control to the handler. CIO also acts as a common exit routine for all of the device handlers. CIO is the common entry point for most of the O.S. I/O functions. For example, BASIC will call CIO while executing an OPEN statement. CIO supports the following functions:

OPEN	Device/file open
CLOSE	Device/file close
GET CHARS	Read 1-N characters
READ RECORD	Read next record
PUT CHARS	Write 1-N characters
WRITE RECORD	Write next record
STATUS	Get device status
SPECIAL	Device handler specific (e.g. NOTE for FMS)

You may wish to make your own CIO calls. The calling sequence for CIO is:

```
                                ;rem user has set up IOCB
        LDX    #IOCBNUM         ;set the IOCB index (IOCB * 16)
        JSR    CIOV             ;system routine vector to CIO
        BMI    ERROR            ;if branch taken then CIO returned
                                ;error code in Y register
```

As shown in the above call, one of the IOCBs is used to communicate control data to CIO. You may use any one of the 8 IOCBs. CIO expects the IOCB index to be in the X register. On return, the status bits of the 6502 are set to indicate success or error in the I/O operation. If the N bit is clear (B'0') the I/O was done successfully, and the Y register will contain a 1. If the N bit is set (B'1'), the I/O request resulted in an error; the Y register will contain the error code. The error/success value is also returned in the IOCB byte ICSTA (see IOCB definition). Chapter 5 of the O.S. User Manual has a sample program segment that calls CIO to OPEN a disk file, READ some records, and CLOSE the file.

CIO routes I/O control data by using the IOCB index in the X register to move the contents of the your IOCB to the ZIOCB. CIO then calculates the device handler entry point and vectors to the appropriate device handler routine. Appendix VI is a flowchart of the CIO system routine.

CIO calculates the device handler entry point in an indirect manner. During an OPEN call, CIO gets the device specification for the file to be OPENed. Let us assume that the device we want to OPEN is the printer. The device specification for the printer would be 'P:'.

```
               01 ;  HANDLER ADDRESS TABLE
E430           02 PRINTV   =      $E430
E440           03 CASETV   =      $E440
E400           04 EDITRV   =      $E400
E410           05 SCRENV   =      $E410
E420           06 KEYBDV   =      $E420
               07 ;
0000           08            *=   $031A
               09 ;
               10 HATABS
031A 50        20            .BYTE   'P'     PRINTER
031B 30E4      30            .WORD   PRINTV     ENTRY POINT TABLE
031D 43        40            .BYTE   'C'     CASSETTE
031E 40E4      50            .WORD   CASETV     ENTRY POINT TABLE
0320 45        60            .BYTE   'E'     DISPLAY EDITOR
0321 00E4      70            .WORD   EDITRV     ENTRY POINT TABLE
0323 53        80            .BYTE   'S'     SCREEN HANDLER
0324 10E4      90            .WORD   SCRENV     ENTRY POINT TABLE
0326 4B        0100          .BYTE   'K'     KEYBOARD
0327 20E4      0110          .WORD   KEYBDV     ENTRY POINT TABLE
0329 00        0120          .BYTE   0       FREE ENTRY 1 (DOS)
032A 00 00     0130          .BYTE   0,0
032C 00        0140          .BYTE   0       FREE ENTRY 2 (850 MODULE)
032D 00 00     0150          .BYTE   0,0
032F 00        0160          .BYTE   0       FREE ENTRY 3
0330 00 00     0170          .BYTE   0,0
0332 00        0180          .BYTE   0       FREE ENTRY 4
0333 00 00     0190          .BYTE   0,0
0335 00        0200          .BYTE   0       FREE ENTRY 5
0336 00 00     0210          .BYTE   0,0
0338 00        0220          .BYTE   0       FREE ENTRY 6
0339 00 00     0230          .BYTE   0,0
033B 00        0240          .BYTE   0       FREE ENTRY 7
```

FIGURE 8.4A HANDLER ADDRESS TABLE (HATABS)

```
        *=$PRINTV
E430 9E EE      .WORD   PHOPEN-1        DEVICE OPEN
E432 DB EE      .WORD   PHCLOS-1        DEVICE CLOSE
E434 9D EE      .WORD   BADST-1         DEVICE READ-NOT IMPLEMENTED
E436 A6 EE      .WORD   PHWRIT-1        DEVICE WRITE
E438 80 EE      .WORD   PHSTAT-1        DEVICE STATUS
E43A 9D EE      .WORD   BADST-1         SPECIAL-NOT IMPLEMENTED
E43C 4C 78 EE    JMP    PHINIT          DEVICE INITIALIZATION
```

FIGURE 8.4B PRINTER HANDLER ENTRY POINT TABLE

CIO uses a table called HATABS (figure 8.4A) to indirectly calculate the handler entry point. This table uses a device specification as a key to find the address of the associated handler entry points. The device specification is used to find the HATABS entry for the device we want to OPEN. CIO starts at the end of HATABS and looks for the first entry that matches the current device specification (in our case 'P:', the first entry in the HATABS). The address associated with the device spec is a pointer to a list of handler entry points (the handler entry points for the printer are shown in figure 8.4B).

CIO uses ICCOM, the IOCB command byte, to find which one of the handler entry points to vector thru. The entry point tables for all of the resident device handlers can be found in the O.S. listing. The positions in all the device handler entry point tables have the same meaning. For example, the first position in all of the device handler entry point tables is the vector to the device handler OPEN routine.

HATABS is located in RAM at $031A and it contains room for 14 entries. At power-up and SYSTEM RESET the contents of HATABS is initialized as shown in figure 8.4. HATABS at this time has entries for all the resident handlers except the disk handler. The disk handler is never called via HATABS (see Chapter 9).

Other entries may be added to HATABS by you or by the O.S. The O.S. may add entries as part of its power-up or SYSTEM RESET function. Any new entries would start at $0329. Additions to HATABS might be entries for the disk drive (the File Manager or FMS) or the 850 module.

You can take advantage of the flexible nature of HATABS to add some new features to the O.S. One example (figure 8.5) shows how to add a null handler. A null handler is exactly what it sounds like, it performs NO FUNCTION! A null handler may be used to debug programs that use devices that have slow access speed. Instead of waiting for 50,000 disk accesses to find a bug, just run the output thru the null handler!

Another use of HATABS is to change the function of an old entry in HATABS. Suppose you wanted to add a printer to your computer that had some special features not supported by the current printer handler. By changing the HATABS entry point table pointer, you can point all 'P:' I/O to your own printer handler. Appendix VII gives an example of a Qume printer handler that uses the front controller ports for speedy transfer of data to be printed.

```
0000         10             *=    $600
031A         20 HATABS      =     $031A
             30 START
0600 A000    40             LDY      #0
             50 LOOP
0602 B91A03  60             LDA      HATABS,Y
0605 C900    70             CMP      #0             FREE ENTRY?
0607 F009    80             BEQ      FOUND
0609 C8      90             INY
060A C8      0100           INY
060B C8      0110           INY                     POINT TO NEXT HATABS ENTRY
060C C022    0120           CPY      #34            AT END OF HATABS?
060E D0F2    0130           BNE      LOOP           NO ...
0610 38      0140           SEC                     YES, FLAG ERROR TO CALLER
0611 60      0150           RTS
             0160 ;
             0170 FOUND
0612 A94E    0180           LDA      #'N            DEVICE NAME
0614 991A03  0190           STA      HATABS,Y
061/ C8      0200           INY
0618 A924    0210           LDA      #NULLTAB&255
061A 991A03  0220           STA      HATABS,Y       NULL HANDLER ENTRY TABLE
061D C8      0230           INY
061E A906    0240           LDA      #NULLTAB/256
0620 991A03  0250           STA      HATABS,Y
0623 60      0260           RTS
             0270 ;
             0280 NULLTAB
0624 3206    0290           .WORD    RTHAND-1       OPEN
0626 3206    0300           .WORD    RTHAND-1       CLOSE
0628 3406    0310           .WORD    NOFUNC-1       READ
062A 3206    0320           .WORD    RTHAND-1       WRITE
062C 3206    0330           .WORD    RTHAND-1       STATUS
062E 3406    0340           .WORD    NOFUNC-1       SPECIAL
0630 4C3306  0350           JMP      RTHAND       INITIALIZATION
             0360 ;
             0370 RTHAND
0633 A001    0380           LDY      #1             SUCCESSFUL I/O FUNCTION
             0390 NOFUNC                            FUNCTION NOT IMPLEMENTED
0635 60      0400           RTS
```

FIGURE 8.5 NULL HANDLER

DOING CIO FROM BASIC

Most of the CIO functions (OPEN, CLOSE, etc.) are available through calls from BASIC. BASIC lacks one set of the functions of CIO, the ability to do non-record I/O more than a byte at a time (GETCHRS and PUTCHRS).

The ability to input or output a buffer of characters at a time is a nifty feature. For instance, an assembly language routine can be loaded directly into memory from a disk file. In a BASIC program, an assembly language routine is normally read into a string and a USR call is made to ADR(string). Since the address of a BASIC string may shift during program modification, the assembly language routine must be location independent. This means memory reference instructions to addresses within the string will not work.

The subroutine in figure 8.6 avoids the use of strings. This way the assembly module does not have to be location independent. Control data is POKEd into an IOCB to read an assembly language routine directly into RAM at the address it was assembled. The BASIC subroutine in figure 8.6 can also be used to output data directly from memory with the user specifing both the location and the length of the data buffer.

THE DEVICE HANDLERS

The device handlers can be divided into the resident and non-resident handlers. The resident handlers are present in the O.S. ROM. Any resident handlers can be called through CIO as long as the handler has an entry in HATABS. The resident device handlers are:

(E:)	DISPLAY EDITOR
(S:)	SCREEN
(K:)	KEYBOARD
(P:)	PRINTER
(C:)	CASSETTE

```
30 REM THIS PROGRAM LOADS PAGE 6 FROM THE FILE D:TEST
100 DIM FILE$(20),CIO$(7):CIO$="hhh*LVd"  (   indicates inverse video)
106 REM CIO$ IS PLA,PLA,PLA,TAX,JMP $E456 (CIOV)
110 FILE$="D:TEST":REM _  *FILE NAME
120 CMD=7:STADR=1536:GOSUB 30000
130 IF ERROR<>1 THEN ? "ERROR - ";ERROR;" AT LINE ";
135 ? PEEK(186)+PEEK(187)*256,PEEK(195)
200 END
300 REM _            CIO SETUP SUBROUTINE
310 REM

30000 REM ROUTINE BY M. EKBERG FOR ATARI 9-3-80
30001 REM
30002 REM THIS ROUTINE LOADS OR SAVES MEMORY FILE FROM BASIC
30003 REM BY SETTING UP AN IOCB AND CALLING CIO DIRECTLY
30004 REM
30006 REM ON ENTRY CMD=7 MEANS LOAD MEMORY
30008 REM _        CMD=11 MEANS SAVE MEMORY
30009 REM _        STADR= THE ADDRESS TO LOAD OR SAVE MEMORY
30010 REM _        BYTES= THE NUMBER OF BYTES TO SAVE OR LOAD
30011 REM _        IOCB= THE IOCB TO USE
30012 REM _        FILE$= DESTINATION FILE NAME
30013 REM _
30014 REM ON EXIT ERROR=1 MEANS SUCCESSFUL COMMAND
30018 REM _       ERROR<>1 THEN ITS AN ERROR STATUS
30019 REM
30020 REM _   *** IOCB EQUATES ***
30022 REM
30024 IOCBX=IOCB*16:ICCOM=834+IOCBX:ICSTA=835+IOCBX
30026 ICBAL=836+IOCBX:ICBAH=837+IOCBX
30028 ICBLL=840+IOCBX:ICBLH=841+IOCBX
30029 REM
30030 AUX1=4:IF CMD=11 THEN AUX1=8
30035 TRAP 30900:OPEN #IOCB,AUX1,0,FILE$:IOCBX=IOCB*16
30040 TEMP=STADR:GOSUB 30500
30090 POKE ICBAL,LOW:POKE ICBAH,HIGH
30100 TEMP=BYTES:GOSUB 30500
30130 POKE ICBLL,LOW:POKE ICBLH,HIGH
30140 POKE ICCOM,CMD:ERROR=USR(ADR(CIO$),IOCBX)
30150 ERROR=PEEK(ICSAT):RETURN
30200 REM
30300 REM _   ***ROUTINE RETURNS HIGH,LOW BYTE OF 16 BIT NUMBER
30400 REM
30500 HIGH=INT(TEMP/256):LOW=INT(TEMP-HIGH*256):RETURN
30550 REM
30600 REM ***TRAP HERE IF ERROR IN ROUTINE****
30900 ERROR=PEEK(195)
30920 CLOSE #IOCB:RETURN
```

FIGURE 8.6 BASIC DIRECT CIO CALL

The non-resident handlers are not present in the O.S. ROM. Non-resident handlers may be added by the O.S. during power-up or SYSTEM RESET, or you can add your own device handler during program execution. Refer to figure 8.5 as an example of adding a handler to the O.S.

The device handlers use I/O control data passed by CIO in the ZIOCB. The data in the ZIOCB is used to perform I/O functions such as OPEN, CLOSE, PUT, and GET. Not all of the device handlers support all the I/O commands (e.g. trying to PUT a character to the keyboard results in an Error 146, Function Not Implemented). Section 5 of the O.S. Manual contains a list of the functions supported by each device handler.

SERIAL I/O SYSTEM ROUTINE - - SIO

SIO AND THE DEVICE HANDLERS

SIO handles serial bus communication between the serial device handlers in the computer and the serial bus devices. SIO communicates to its caller through the device control block (DCB). SIO uses the I/O control data in the DCB to send and receive commands and data over the serial bus. The calling sequence is:

```
                        ;caller has set up the DCB to do function
        JSR SIOV        ;system vector to SIO
        BMI ERROR       ;N bit set indicates error in I/O execution
```

The DCB contains I/O control information for SIO and must be setup before the call to SIO. Figure 8.3 shows the contents of the DCB for some common I/O operations.

You need to understand the structure of the DCB to send commands to SIO. The DCB is described in section 9 of the O.S. Manual. Figure 8.7 demonstrates a simple assembly language routine to output a line to the printer by setting up the DCB and calling SIO.

```
0000           05                *= $3000       ARBITRARY START
               10 ;THIS ROUTINE PRINTS A LINE TO THE PRINTER BY CALLING SIO
E459           20 SIOV        =        $E459   SIO VECTOR
009B           30 CR          =        $9B     EOL
0040           40 PRNTID      =        $40     PRINTER SERIAL BUS ID
004E           45 MODE        =       $4E      NORMAL MODE
001C           50 PTIMOT      =        $001C   TIMEOUT LOCATION
0300           60 DDEVIC      =        $300    DEVICE SERIAL BUS ID
0301           70 DUNIT       =        $301    SERIAL UNIT NUMBER
0302           80 DCOMND      =        $302    SIO COMMAND
0303           90 DSTATS      =        $303    SIO DATA DIRECTION
0304           0100 DBUFLO    =        $304    BUFFER LOW ADDRESS
0305           0110 DBUFHI    =        $305    BUFFER HIGH ADDRESS
0306           0120 DTIMLO    =        $306    SIO TIMEOUT
0307           0130 DTIMHI    =        $307
0308           0140 DBYTLO    =        $308    BUFFER LENGTH
0309           0150 DBYTHI    =        $309
030A           0160 DAUX1     =        $30A    AUXILARY BYTE---PRINTER MODE
030B           0170 DAUX2     =        $30B    AUXILARY BYTE---NOT USED
               0180 ;
3000 455841 0190 MESS      .BYTE    "EXAMPLE 12",CR
3001 4D504C
3005 452031
3009 329B
            0200 ;
300B A940   0220           LDA      #PRNTID SET BUS ID
300D 8D0003 0230           STA      DDEVIC
3010 A901   0240           LDA      #1      SET UNIT NUMBER
3012 8D0103 0250           STA      DUNIT
3015 A94E   0260           LDA      #MODE
3017 8D0A03 0270           STA      DAUX1   PRINTER MODE NORMAL
301A A901   0275           LDA      #1
301C 8D0B03 0280           STA      DAUX2   UNUSED
301F 8D0703 0290           STA      DTIMHI  TIMEOUT<256 SECS
3022 A51C   0300           LDA      PTIMOT  SET SIO TIMEOUT FOR PRINTER
3024 8D0603 0310           STA      DTIMLO
3027 A900   0320           LDA      #MESS&255
3029 8D0403 0330           STA      DBUFLO  SET MESS AS BUFFER
302C A930   0340           LDA      #MESS/256
302E 8D0503 0350           STA      DBUFHI
3031 A980   0360           LDA      #$80    SET SIO DATA DIRECTION FOR
3033 8D0303 0370           STA      DSTATS     PERIPHERAL TO RECEIVE
3036 A957   0380           LDA      #'W     SIO COMMAND WRITE
3038 8D0203 0390           STA      DCOMND
303B 2059E4 0410           JSR      SIOV    CALL SIO
303E 3001   0420           BMI      ERROR
3040 00     0430 GOOD      BRK
3041 00     0440 ERROR     BRK
```

FIGURE 8.7 SIO CALL TO DUMP LINE TO PRINTER

SIO INTERRUPTS

SIO uses three IRQ interrupts to send and receive serial bus communications to serial bus devices. They are:

IRQ	Location, Length	Function
VSERIR	[$020A,2]	SERIAL INPUT READY
VSEROR	[$020C,2]	SERIAL OUTPUT NEEDED
VSEROC	[$020E,2]	TRANSMISSION FINISHED

All program execution is halted while SIO uses the serial bus for communication. For output, SIO gives the POKEY serial output shift register (SEROUT) a byte to send on the serial bus, then it enters an idle loop until it receives an IRQ (VSEROR) from POKEY telling SIO that SEROUT is free for another byte. The IRQ causes a jump to the SIO routine to load SEROUT with another byte. This loop is executed until all the bytes specified in the DCB buffer length have been sent. VSEROR is an IRQ indicating that the transmission of the byte over the bus is complete.

The SIO execution for input is similar. POKEY informs SIO a byte has been received in the serial input shift register (SERIN) by generating an IRQ (VSERIR). SIO stores the byte in a buffer and then SIO idles in a loop until the next IRQ tells it another byte has been received.

You may have noticed from the above explanation that SIO wastes some time idling while waiting for POKEY to send or receive information on the bus. As the vectors for the three SIO IRQ service routines are RAM vectors, they can be used by some handlers to improve system I/O performance.

Indeed, this is how the 850 module is able to do 'concurrent I/O'. The 850 module handler takes over the SIO IRQ vectors and points these IRQ vectors to the module's own IRQ routines while in concurrent I/O. The 850 Module handler can then send commands over the bus. While waiting for the 850 Module to do the command, the 850 Module handler allows the executing program to continue.

SECTION 8.3 - - INTERRUPTS

INTRODUCTION

In the previous section we have seen how SIO uses interrupts to coordinate transfer of data over the serial bus. The computer also has other catagories of interrupts. You can use several of these other interrupts to add some powerful features to your application. It should be noted that there are two types of interrupts, the maskable (IRQ) and the non-maskable (NMI) interrupts. The PCS interrupts are:

Name (vector)	Type	Function	Used By
DISPLAY LIST (VDSLST)	NMI	Graphics timing	User
SYSTEM RESET (none)	NMI	System init.	PCS
VERTICAL BLANK (VVBLKI,VVBLKD)	NMI	Graphics display	PCS,user
SERIAL OUTPUT READY (VSERIN)	IRQ	Serial input	PCS
SERIAL INPUT READY (VSEROR)	IRQ	Serial input	PCS
SERIAL OUTPUT COMPLETE (VSEROC)	IRQ	Serial output	PCS
POKEY TIMER 1 (VTIMR1)	IRQ	Hardware timer	User
POKEY TIMER 2 (VTIMR2)	IRQ	Hardware timer	User
*POKEY TIMER 4 (VTIMR4)	IRQ	Hardware timer	User
KEYBOARD (VKEYBD)	IRQ	Key hit	PCS
BREAK KEY (none)	IRQ	<BREAK> hit	PCS
SERIAL BUS PROCEED (VPRCED)	IRQ	Device proceed	Unused
SERIAL BUS INTERRUPT (VINTER)	IRQ	Device interrupt	Unused

* This IRQ is not vectored in the current O.S.

If you are not familiar with interrupts, Section 6 of the O.S. Manual contains information on them. Working with interrupts can be tricky. For example, if you accidentally disable the keyboard IRQ interrupt, the computer will ignore all the keys except the <BREAK> key. Although this may be useful sometimes, it may make debugging your program a bit difficult!

THE IRQ INTERRUPT HANDLER

The O.S. has an IRQ interrupt handler that processes the various IRQs. The IRQ Handler has RAM vectors for all of the IRQs except the <BREAK> key IRQ. The IRQ vectors are set to their initial values during both power-up and SYSTEM RESET. The locations of the IRQ RAM vectors are described in figure 8.9B.

The IRQs vectors are:

VIMIRQ -- Immediate IRQ vector. All IRQ's vector thru this location. VIMIRQ normally points to the IRQ Handler. You can 'steal' this vector to do your own IRQ interrupt processing.

VSEROR -- Pokey Serial Output Ready IRQ vector. (see section 8.2)

VSERIN -- Pokey Serial Input Ready IRQ vector. (see section 8.2)

VSEROC -- Pokey Serial Output Complete IRQ vector. (see section 8.2)

VTIMR1 -- Pokey Timer 1 IRQ vector. (see section 8.7 for information on the Pokey Timers)

VTIMR2 -- Pokey Timer 2 IRQ vector.

VTIMR4 -- Pokey Timer 4 IRQ vector.

VKEYBD -- Keyboard IRQ vector. Pressing any key except <BREAK> causes this IRQ. VKEYBD can be used to pre-process the key code before it is converted to ATASCII (by the O.S.). VKEYBD normally points to the O.S. keyboard IRQ routine.

VPRCED -- Peripheral Proceed IRQ vector. The proceed line is available to peripherals on the serial bus. This IRQ is unused at present and normally points to an RTI.

VINTER -- Peripheral interrupt IRQ vector. The interrupt line is also available on the serial bus. VINTER also normally points to an RTI.

VBREAK -- 6502 BRK instruction IRQ vector. Whenever a $00 opcode (the software break instruction) is executed, this interrupt occurs. VBREAK can be used to set break points for a debugger. VBREAK normally points to an RTI.

The IRQs are enabled and disabled as a group by the 6502 instructions CLI and SEI respectively. The IRQs also have individual enable/disable bits. Section III of the hardware manual shows the IRQs and their enable/disable bits.

Register (IRQEN) contains most of the IRQ enable/disable bits and is a write-only register. The O.S. keeps a shadow of IRQEN for you to read in POKMSK. POKMSK is written to IRQEN during vertical blank.

USING THE IRQS

Many applications require that the keyboard be 'idiot-proofed'. This means a user can press any key or combination of keys and the program will accept valid key sequences and ignore invalid ones. You can use a couple of the IRQ vectors to 'idiot-proof' your program. The example in figure 8.8 uses the VKEYBD IRQ vector to disable the control key. The routine also masks the <BREAK> key by stealing the VIMIRQ vector and ignoring the <BREAK> key interrupt.

THE NMI HANDLER

The O.S. has an NMI handler for handling the non-maskable interrupts. Unlike the IRQs, the NMIs cannot be 'masked' (disabled) on the 6502. All the NMIs except SYSTEM RESET can be disabled on ANTIC.

Two of the NMIs, the display list interrupt (DLI) and the vertical blank (VBLANK) interrupt, have RAM vectors that you can use. In fact, VBLANK can be intercepted in two places, Immediate or Deferred VBLANK. The NMI vectors are:

Name	Vector
SYSTEM RESET	none
DISPLAY LIST INTERRUPT	VDSLST [$0200]
VERTICAL BLANK	
IMMEDIATE	VVBLKI [$0222]
DEFERRED	VVBLKD [$0224]

The SYSTEM RESET NMI doesn't have a RAM vector. SYSTEM RESET always results in a jump to the monitor warmstart routine (section 8.5) The DLI and the VBLANK interrupts are covered in Chapter 5 and Appendix I respectively.

```
0010        10 POKMSK   =    $0010
D209        20 KBCODE   =    $D209
0208        30 VKEYBD   =    $0208
D20E        40 IRQEN    =    $D20E
D20E        45 IRQST    =    IRQEN
0216        46 VMIRQ    =    $0216
0000        60          *=   $600
0600 78     80 START    SEI                DISABLE IRQS
0601 AD1602 90          LDA      VMIRQ     REPLACE THE IRQ VECTOR
0604 8D4D06 0100        STA     NBRK+1         WITH OUR OWN
0607 AD1702 0110        LDA     VMIRQ+1        ALL IRQS WILL
060A 8D4E06 0120        STA     NBRK+2         GO TO NBRK
060D A945   0130        LDA     #IRQ&255
060F 8D1602 0140        STA     VMIRQ
0612 A906   0150        LDA     #IRQ/256
0614 8D1702 0160        STA     VMIRQ+1
0617 58     0170        CLI                ENABLE IRQS
0618 AD0802 0200        LDA     VKEYBD     POINT KEY IRQ TO
061C 8D4306 0210        STA     JUMP+1     REP
061F AD0902 0220        LDA     VKEYBD+1
0622 8D4406 0230        STA     JUMP+2
0625 A939   0240        LDA     #REP&255          VECTOR KEY IRQ
0627 8D0802 0250        STA     VKEYBD            LOW BYTE OF VECTOR
062A A906   0260        LDA     #REP/256
062C 8D0902 0270        STA     VKEYBD+1
0638 60     0280        RTS
            0290        *=$639
0639 AD09D2 0300 REP    LDA     KBCODE     ALL KEY IRQS COME HERE
063C 2980   0310        AND     #$80       CHECK IF CONTROL HIT
063E F002   0320        BEQ     JUMP       IF NOT HIT THEN GO
0640 68     0330        PLA                ELSE IGNORE CON. KEY
0641 40     0340        RTI
0642 4C4206 0360 JUMP   JMP     JUMP       THIS CALLS THE OLD KEY IRQ
0645 48     0375 IRQ    PHA                ALL IRQS COME HERE
0646 AD0ED2 0380        LDA    IRQST       CHECK IF <BREAK>
0649 1004   0390        BPL    BREAK       IF <BREAK> IRQ BR.
064B 68     0405        PLA                ELSE CALL OLD IRQ VECTOR
064C 4C4C06 0410 NBRK   JMP    NBRK        CALL OLD IRQ VECTOR
064F A97F   0430 BREAK  LDA    #$7F        HERE IF <BREAK>
0651 8D0ED2 0440        STA    IRQST       SHOW NO <BREAK>
0654 A510   0450        LDA    POKMSK
0656 8D0ED2 0460        STA    IRQEN
0659 68     0462        PLA
065A 40     0464        RTI                RETURN AS IF NO <BREAK>
065B        0470        *=   $02E2
02E2 0006   0480        .WORD START
```

FIGURE 8.8 IDIOT-PROOFING THE KEYBOARD

SECTION 8.4 - - THE SYSTEM VECTORS

The O.S. has two types of vectors, ROM vectors and RAM vectors. ROM vectors are locations which contain JMP instructions to system routines. RAM vector contain two-byte addresses to system routines, handler entry pointers (see CIO section 8.2), or to initialization routines.

Both the ROM and RAM vector addresses are guaranteed not to change if ATARI releases a new O.S. but the contents of these vectors are free to change. The vectors, their contents and a brief description of their function are described in figures 8.9A and 8.9B.

ROM VECTORS

Name	Location	Use
DISKIV	$E450	Disk handler intialization
DSKINV	$E453	Disk handler vector
CIOV	$E456	Central input/output routine vector
SIOV	$E459	Serial input/output routine vector
SETVBV	$E45C	Set system timers routine vector
SYSVBV	$E45F	System vertical blank calculations
XITVBV	$E462	Exit vertical blank calculations
SIOINV	$E465	Serial input/output initialization
SENDEV	$E468	Serial bus send enable routine
INTINV	$E46B	Interrupt handler routine
CIOINV	$E46E	Central input/output initialization
BLKBDV	$E471	Blackboard mode (memopad) vector
WARMSV	$E474	Warm start entry point (SYSTEM RESET)
COLDSV	$E477	Cold start entry point (power-up)
RBLOKV	$E47A	Cassette read block routine vector
CSOPIV	$E47D	Cassette open for input vector

An example of using a ROM vector is:

```
JSR CIOV
```

FIGURE 8.9A ROM VECTORS

RAM VECTORS

Name	Location	Value	Use
VDSLST	$0200	$E7B3	Display List Interrupt NMI Vector
VPRCED	$0202	$E7B3	Proceed Line IRQ Vector -- Unused at present
VINTER	$0204	$E7B3	Interrupt Line IRQ Vector -- Unused at Present
VBREAK	$0206	$E7B3	Software Break Instuction IRQ Vector
VKEYBD	$0208	$FFBE	Keyboard IRQ Vector
VSERIN	$020A	$EB11	Serial Input Ready IRQ Vector
VSEROR	$020C	$EA90	Serial OutputReady IRQ Vector
VSEROC	$020E	$EAD1	Serial Output Complete IRQ Vector
VTIMR1	$0210	$E7B3	POKEY Timer 1 IRQ Vector
VTIMR2	$0212	$E7B3	POKEY Timer 2 IRQ Vector
VTIMR4	$0214	$E7B3	POKEY Timer 4 IRQ Vector
VIMIRQ	$0216	$E6F6	Immediate IRQ Vector to IRQ Handler
VVBLKI	$0222	$E7D1	Vertical Blank NMI Vector
VVBLKD	$0224	$E93E	Deferred Vertical Blank Vector
CDTMA1	$0226	$xxxx	System Timer 1 JSR Address
CDTMA2	$0228	$xxxx	System Timer 2 JSR Address
CASINI	$0002	$xxxx	Vector for bootable cassete program initialization
DOSINI	$000C	$xxxx	Disk Initilization Vector
DOSVEC	$000A	$xxxx	Disk Software Run Vector
RUNVEC	$02E0	$xxxx	DUP File Load and GO Run Vector
INIVEC	$02E2	$xxxx	DUP File Load and Go Initialization Vector
HATABS	$031A	'P'	Printer Device I.D
	$031B	$E430	Address of Printer Entry Points Table
	$031D	'C'	Cassette Device I.D.
	$031E	$E440	Address of Cassette Entry Points Table
	$0320	'E'	Display Editor I.D.
	$0321	$E400	Address of Display Editor Entry Points Table
	$0323	'S'	Screen Handler I.D.
	$0324	$E410	Address of Screen Display Entry Point Table
	$0326	'K'	Keyboard Handler I.D.
	$0327	$E420	Address of Keyboard Handler Entry Point Table
	$0329	'x'	HATABS Unused Entry 1
	$032B	'x'	" " " 2
	$032E	'x'	" " " 3
	$0331	'x'	" " " 4
	$0234	'x'	" " " 5
	$0237	'x'	" " " 6
	$032A	'x'	" " " 7
	$034D	'x'	" " " 8
	$0340	'x'	" " " 9

An x indicates content varies.
An example of using a RAM vector is:

```
      JSR CALL
CALL  JMP (DOSINI)
```

FIGURE 8.9B RAM VECTORS

[THIS PAGE INTENTIONALLY LEFT BLANK]

```
             0010 ; WRITTEN BY...MICHAEL EKBERG
0600         0030 START    =    $600
000C         0040 DOSINI   =    $0C
02E7         0050 MEMLO    =    $2E7
3000         0060 NEWMEM   =    $3000     ; ALTER THIS TO GET SIZE
             0070 ; THIS ROUTINE RESERVES SPACE FOR ASSEMBLY ROUTINES
             0090 ; BY RESETING THE MEMLO POINTER. IT RUNS AS
             0100 ; AN AUTORUN.SYS FILE. IT ALSO RESETS MEMLO ON [RESET].
             0120 ; MEMLO IS SET TO THE VALUE OF NEWMEM.
             0130 ;
             0140 ; THIS PART IS PERMANENT, IE. NEEDS TO BE RESIDENT.
             0150 ; THE SYSTEM DOS INIT VECTOR HAS BEEN STOLEN
             0160 ; AND STORED IN THE LOCATION INITDOS+1&2.
             0180 ; DOS IS INITIALIZED AND MEMLO IS INITIALIZED
             0190 ; INITDOS EXECUTES ON [RESET].
0000         0200        *=   START
             0210 INITDOS
0600 200D06  0220        JSR  CYNTHIA   ; DO DOS INITLIST
0603 A900    0230        LDA  #NEWMEM&255
0605 8DE702  0240        STA  MEMLO
0608 A930    0250        LDA  #NEWMEM/256
060A 8DE802  0260        STA  MEMLO+1
             0270 CYNTHIA
060D 60      0280        RTS
             0290 ; THIS PART IS EXECUTED AT POWER UP ONLY AND
             0300 ; CAN BE DELETED AFTER POWER-UP.
             0330 ; THIS ROUTINE STORES THE CONTENTS OF DOSINI INTO A JSR
             0350 ; AT LOCATION INITDOS+1. IT THEN REPLACES DOSINI WITH
             0370 ; IT'S OWN VALUE, THE LOCATION INITDOS.
             0390 JACKIE
060E A50C    0400        LDA  DOSINI    ; SAVE DOSINI
0610 8D0106  0410        STA  INITDOS+1
0613 A50D    0420        LDA  DOSINI+1
0615 8D0206  0430        STA  INITDOS+2
0618 A900    0440        LDA  #INITDOS&255  ; SET DOSINI
061A 850C    0450        STA  DOSINI
061C A906    0460        LDA  #INITDOS/256
061E 850D    0470        STA  DOSINI+1
0620 A500    0480        LDA  NEWMEM&255  ; SET MEMLO
0622 8DE702  0490        STA  MEMLO
0625 A930    0500        LDA  #NEWMEM/256
0627 8DE802  0510        STA  MEMLO+1
062A 60      0520        RTS
062B         0530        *=   $2E2
02E2 0E06    0540        .WORD JACKIE   ; SET RUN ADDRESS
02E4         0550        .END
```

FIGURE 8.10 MEMLO MOVER

SECTION 8.5 - - THE MONITOR

The O.S. monitor is a program in ROM that handles both the system power-up and SYSTEM RESET sequences. The power-up and SYSTEM RESET sequences are similar in function and in fact share much of the same code. Appendix IV has a flowchart of the power-up and SYSTEM RESET routines.

The power-up routine (also known as coldstart) is invoked either by turning on the computer or by jumping to COLDSV [$E477]. COLDSV is the system routine vector to the power-up routine. Important items to remember about the power-up are:

1. ALL of memory is cleared except locations $0000-$000F

2. Both a cassette and disk boot are attempted. BOOT?[$0009] is a flag that indicates the success or failure of the boots. Bit 0 = 1 if a successful cassette boot; Bit 1 = 1 if a successful disk boot.

3. COLDST[$0244] is a flag that tells the Monitor that it is in the middle of power-up. COLDST=0 means System Reset, COLDST<>0 means power-up. An interesting use of COLDST is to set it to a non-zero value during execution of an application program. This will cause a SYSTEM RESET to become a power-up. This technique may add a measure of security by preventing a user from gaining control of the computer while an application is running.

Pressing the <SYSTEM RESET> key causes a SYSTEM RESET (also known as warmstart). Some of the key facts to remember about SYSTEM RESET in the flowchart in Appendix IV are:

1. The O.S. RAM vectors are downloaded from ROM both during SYSTEM RESET and power-up. If you wish to 'steal' a vector, some provision must be made to handle SYSTEM RESET. See Chapter 9 on how to handle SYTEM RESET.

2. MEMLO,MEMTOP,APPMHI,RAMSIZ, and RAMTOP are reset during System Reset. If you wish to alter these user RAM pointers to reserve some space for assembler modules called by BASIC, some provision must be made to handle SYSTEM RESET. Figure 8.10 is an example on how to do this.

SECTION 8.6 - - THE SYSTEM DATABASE

The system database contains many locations that store information of interest to the user. The locations may be accessed directly by Assembly language programs or by PEEK and POKE in BASIC.

The system database occupies RAM pages 0 thru 4 ($0000-$03FF). The database contains system flags, I/O buffers, I/O parameters, etc. The user can use some of these locations to provide features not supplied by either the O.S. or a program environment (i.e. BASIC). Figure 8.11 describes some of these elements of the database.

DATA BASE CHART

	Location			
Name	,Size	Init. by	Value	Function
MEMLO	$02E7,2	R,P	x	User Free Memory Low Address
MEMTOP	$02E5,2	R,P	x	User Free Memory High Address
APPMHI	$000E	P,R	$00	User Free Memory Screen lower limit
RAMTOP	$006A,2	P,R	x	Display Handler Top of RAM Address(msb)
RAMSIZ	$02E4,1	P,R	x	Top of RAM Address (msb)
POKMSK	$0010,1	P,R	x	O.S. IRQ enable shadow
BRKKEY	$0111,1	P,R	$FF	BREAK key flag; $FF means no BREAK key hit
IRQEN	$D20E	P,R	x	ANTIC IRQ enable bit register
PTIMOT	$001C	P,R	x	Printer timeout value for SIO
CIOCHR	$002F		x	CIO temp storage for PUT a single char function
ZIOCB	$0020,16			CIO ZERO page I/O BLOCK
BOOT?	$0009,1	P,R	x	Boot Flag; bit 0 is cassette, bit 1 is DOS
CKEY	$004A,1	P	x	Monitor Cassette boot flag
CASSBT	$004B	P		Cassette boot flag
KBCODE	$D209,1		x	Keyboard code register
CH	$02FC	P,R	$FF	Current key value
CH1	$02F2,1		x	Last keyboard value
SHLOK	$02BE,1	P,R	$40	Shift lock
KEYDEL	$02F1,1		$03	Keyboard sotware debounce timer
SSFLAG	$02FF,1		x	Start/stop flag for display
ATRACT	$004D,1			ATRACT mode flag
SRTIMR	$022B,1		x	ATRACT mode timeout timer
COLDST	$0244,1			init flag
WARMST	$0008,1			init flag
CHBAS	$02F4,1	P,R	$E0	Pointer to character base
CRITIC	$0042	P,R		Critical I/O region flag

P means power-up
R means SYSTEM RESET
x means value varies

FIGURE 8.11 SYSTEM DATA BASE

MEMORY POINTERS

The O.S. uses five locations to keep track of the user and display memory, MEMLO, MEMTOP, APPMHI, RAMTOP, and RAMSIZ. Their relationships are shown in figure 8.12, a simple system memory map.

MEMLO is a two-byte location that the O.S. uses to indicate where an application program may begin. Used carefully, MEMLO can be used to create areas for assembly language modules that may be called from BASIC. BASIC uses the value of MEMLO to determine the starting location of a program (see the BASIC chapter for the structure of a BASIC program). If we desire to set MEMLO to a higher address, we must set it before the BASIC cartridge is run. MEMLO has to be used carefully because it is reset by both power-up and SYSTEM RESET.

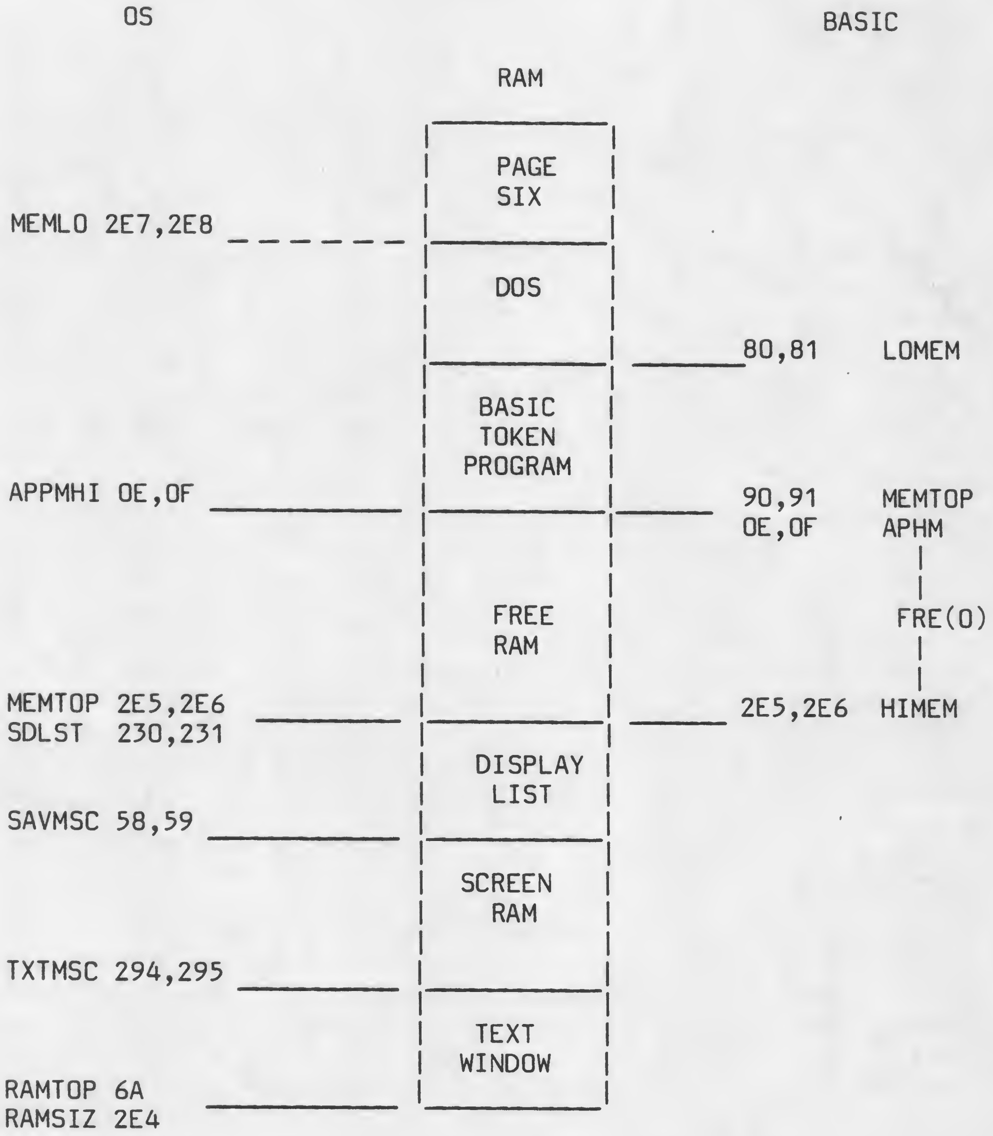
If the system will be booted from a disk drive then the AUTORUN.SYS facility can be used to set MEMLO to a predefined value. Because the DOS also resets MEMLO during SYSTEM RESET via the DOSINI vector, DOSINI needs to 'stolen'. DOSINI contains the address of the DOS initialization code called as part of the monitor system initialization. DOS still needs to be initialized as part of SYSTEM RESET. The contents of DOSINI must be moved into the two-byte address of a JSR instruction as part of the power-up. DOSINI is then set to the address of the JSR instruction for the initialization code and MEMLO is set to the predefined value. When a SYSTEM RESET occurs, the new initialization code is called and the first insruction, JSR OLDDOSINI, initializes DOS. The new initialization code sets MEMLO to the predefined value and RTSs to the old initialization sequence. Figure 8.10 is an example on how to do this.

The above technique can also be used with MEMTOP, the user high memory pointer. Space for assembly modules and data can be set aside by lowering MEMTOP from the values set by power-up and SYSTEM RESET. Using MEMTOP instead of MEMLO creates one problem. MEMTOP fluctuates due to both the amount of RAM in the system and the graphics mode of the display. This makes it difficult to predict its value before actually examining the location unless you make some assumptions about the system. As a result all assembly modules may need to be relocatable.

APPMHI is a location which contains an address that specifies the lowest address the display RAM may be placed. Setting APPMHI insures that the display handler will not clobber some of your program code or data.

RAMSIZ, like MEMTOP, can also be used to reserve space for assembly modules. The advantage of RAMSIZ over MEMTOP is that the space saved by moving RAMSIZ down is above the display. Space saved by moving MEMTOP down remains below the display. The display area expands and contracts depending on the graphics mode.

MEMORY MAP



OS AND BASIC POINTERS (DOS PRESENT)

FIGURE 8.12

MISCELLANEOUS

BRKKEY is a flag that is set when the O.S. senses that the <BREAK> key has been pressed. BRKKEY's normal value is $FF. If it changes, then the <BREAK> key has been pressed. The <BREAK> key IRQ (not the software instruction BRK IRQ) must be enabled for the O.S. to sense the <BREAK> key.

The computer's printer timeout value is stored in a RAM location called PTIMOT. It contains the value of the timeout period for SIO in seconds. It may be altered to increase or decrease the timeout period by placing a new value in PTIMOT. PTIMOT is initialized to 30 seconds, and is updated each time the printer is opened. Typical timeout for the 825 is 5 seconds.

```
1 POKE 752,1:GOTO 3
3 ? "†":REM clear screen
4 ? "HOUR";:INPUT HOUR:? "MINUTE ";:INPUT MIN:? "SECOND";:INPUT SEC
5 CMD=1:GOSUB 45
6 ? "†";HOUR;" : ";MIN;" : ";SEC:? " ":? " "
7 CMD=2:GOSUB 45
9 ? "";HOUR;":";MIN;":";SEC;"   ":GOTO 7

10 REM THIS IS A DEMO OF THE REAL TIME CLOCK
20 REM THIS ROUTINE ACCEPTS AN INITIAL TIME IHOUR,IMIN,ISEC
30 REM IT SETS THE REAL TIME CLOCK TO ZERO
40 REM THE CURRENT VALUE OF RTCLOCK IS USED TO ADD TO THE INITIAL TIME TO GET
THE CURRENT TIME HOUR,MIN,SEC
45 HIGH=1536:MED=1537:LOW=1538
50 REM
60 REM ******ENTRY POINT******
65 REM
70 ON CMD GOTO 100,200
95 REM
96 REM ****INITIALIZE CLOCK*****
97 REM
100 POKE 20,0:POKE 19,0:POKE 18,0
105 DIM CLOCK$(50)
106 CLOCK$=" " :(SUB 300
110 IHOUR=HOUR:IMIN=MIN:ISEC=SEC:RETURN
197 REM
198 REM *******READ CLOCK*****
199 REM
200 REM
201 A=USR(ADR(CLOCK$))
210 TIME=((((PEEK(HIGH)*256)+PEEK(MED))*256)+PEEK(LOW))/59.923334
220 HOUR=INT(TIME/3600):TIME=TIME-(HOUR*3600)
230 MIN=INT(TIME/60):SEC=INT(TIME-(MIN*60))
235 SEC=SEC+ISEC:IF SEC>60 THEN SEC=SEC-60:MIN=MIN+1
236 MIN=MIN+IMIN:IF MIN>60 THEN MIN=MIN-60:HOUR=HOUR+1
237 HOUR=HOUR+IHOUR
240 HOUR=HOUR-(INT(HOUR/24))*24
250 RETURN
```

FIGURE 8.13 - - REAL TIME CLOCKS

```
300 FORI=1 TO 38:READ Z:CLOCK$(I,I)=CHR$(Z):NEXT I:RETURN
310 DATA 104,165,18,141,0,6,165,19,141,1,6,165
320 DATA 20,141,2,6,165,18,205,0,6,208,234
330 DATA 165,19,205,1,6,208,227,165,20,205,2,6,208,220,96
```

SECTION 8.7 - - TIMERS

THE SYSTEM TIMERS

There are two types of timers provided for you to use. System timers run at the frequency of the T.V. frame. For North American television (NTSC) the rate is 59.923334 hz. European (PAL) televisions run at 50 hz. The POKEY timers are clocked by frequencies setable by the user.

There are 6 system timers:

Name	Location	Vector or flag
RTCLOK	[$0012,3]	none
CDTMV1	[$0218,2]	CDTMA1 [$0226,2]
CDTMV2	[$021A,2]	CDTMA2 [$0228,2]
CDTMV3	[$021C,2]	CDTMF3 [$022A,1]
CDTMV4	[$021E,2]	CDTMF4 [$022C,1]
CDTMV5	[$0220,2]	CDTMF5 [$022E,1]

All of the system timers are clocked as part of the vertical blank (VBLANK) process. If the VBLANK process is disabled or intercepted, the timers will not be updated.

The real time clock(RTCLOK) and system timer 1 (CDTMV1) are updated during Immediate VBLANK, Stage 1. RTCLOK counts up from 0 and is a three byte value. When RTCLOK reaches its maximum value (16,777,216) it will be reset to zero. RTCLOK can be used as a time piece as example 8.13 shows.

Because the system timers are updated as part of the VBLANK process, special care is needed to set them correctly. A system routine called SETVBV [$E45C] is used to set them. The call to SETVBV is:

```
REG's:X,Y contain the timer value.
      A   contains the timer number:
                   A=1-5  - - TIMERS 1-5
```

ex.:

```
      LDA #1      ;Set system timer 1
      LDY #0
      LDX #02     ;value is $200 VBLANKS
      JSR SETVBV  ;Call system routine to set timer.
```

```
5 ? "†"          :REM BASIC ROUTINE TO SET THE RATE OF A METRONOME
7 REM BY M.EKBERG FOR CARLA
10 X=10: FOR I=1 TO 2 STEP 0
20 TOP=10: FOR J=1 TO TOP : NEXT J    SOFTWARE DELAY LOOP
50 IF STICK(0)=13 THEN X=X+1 :REM     STICK FORWARD MEANS SPEED UP RATE
51 IF STICK(0)=14 THEN X=X-1 :REM     STICK BACK MEANS SLOW METRONOME RATE
52 IF X<1 THEN X=1:REM                NEVER GO BELOW ZERO
53 IF X>255 THEN X=255:REM             OR ABOVE 255
54 REM                                PRINT BEATS/MINUTE
56 ? "";INT(3600/X);" BEATS/MINUTES       "
60 POKE 0,X:REM                       LOACTION $0000 HOLDS THE RATE FOR
70 NEXT I :REM                        THE FOLLOWING ASSEMBLY ROUTINE
```

```
40              *=$600
50 ;METRONOME ROUTINE...USES $0000 PASS THE METRONOME RATE
60 ;
70 AUDF1     =       $D200   AUDIO FREQUENCY REGISTER
80 AUDC1     =       $D201   AUDIO CONTROL REGISTER
90 FREQ      =       $08     AUDF1 VALUE
0100 VOLUME    =       $AF     AUDC1 VALUE
0110 OFF       =       $A0     TURN OFF VOLUME
0120 SETVBV    =       $E45C   SET TIMER VALUE ROUTINE
0130 XITVBV    =       $E462
0140 CDTMV2    =       $021A   TIMER 2
0150 CDTMA2    =       $0228   TIMER 2 VECTOR
0160 ZTIMER    =       $0000   ZPAGE VBLANK TIMER VALUE
0170 ;
0180 START     LDA     #10
0190           STA     ZTIMER
0200 INIT
0210 ;     SET THE TIMER VECTOR
0220 ;
0230           LDA     #CNTINT&255
0240           STA     CDTMA2
0250           LDA     #CNTINT/256
0260           STA     CDTMA2+1
0270 ;
0280 ;     SET THE TIMER VALUE AFTER THE VECTOR
0290 ;
0300           LDY     ZTIMER  SET TIMER TWO TO COUNT
0310           JSR     SETIME
0320           RTS
```

```
0340 ;      METRONME COUNT DOWN VECTORS TO HERE
0350 ;
0360 CNTINT
0370 ;
0380 ;      SET UP AUDIO CHANNEL FOR MET CLICK
0390 ;
0400          LDA     #VOLUME
0410          STA     AUDC1
0420          LDA     #FREQ
0430          STA     AUDF1
0433          LDY     #$FF     DELAY
0437 DELAY
0440          DEY
0442          BNE     DELAY
0450          STY     AUDC1
0460          JMP     INIT
0470 ;
0480 ;
0490 ;      SUBROUTINE TO SET TIMER
0500 ;
0510 SETIME
0520          LDX     #0                NO TIME >256 VBLANKS
0530          LDA     #2                SET TIMER 2
0540          JSR     SETVBV            SYSTEM ROUTINE TO SET TIMER
0550          RTS
0560          *=$2E2
0570          .WORD   START
0580          .END
```

FIGURE 8.14 - - METRONOME

System timers 1-5 are two byte counters. They may be set to a value using the SETVBV routine. The O.S. then decrements them during VBLANK. Timer 1 is decremented during Immediate VBLANK, Stage 1. Timers 2-5 are decremented during Immediate VBLANK, Stage 2. Different actions are taken by the O.S when the different timers are decremented to 0.

System timers 1 and 2 have vectors associated with them. When timer 1 or 2 reaches 0, the O.S. simulates a JSR thru the vector for the given timer. Figure 8.9B gives the vectors for the two timers.

System timers 3-5 have flags that are normally SET (i.e. non-zero). When either of the three timers count to 0, the O.S. will clear (zero) that timer's flag. The user may then test the flag and take appropriate action.

Timers 1-5 are general purpose software timers that may be used for a variety of applications. For example, timer 1 is used by SIO to time serial bus operations. If the timer counts to zero before a bus operation is complete, a 'timeout' error is returned. Timer 1 is set to various values depending on the device being accessed. This insures that, while a device has ample time to answer an I/O request, the computer will not 'hang-up' indefinitly waiting for a non-existant device to respond. Timer 3 is also used by the O.S. The cassette handler uses timer 3 to set the length of time to read and write tape headers. Example 8.14 uses timer 2 to output a sound used as a metronome. The metronome's rate can be set to various values by using a joystick.

CHAPTER 9
THE DISK OPERATING SYSTEM

INTRODUCTION

The Disk Operating System (DOS) is an extension of the O.S. that allows you to access disk drive mass storage as files. You can access disk files just like other files. Let's discuss the parts of DOS, then we will discuss how to use DOS.

The Disk Operating System (DOS) has three parts, the resident disk handler, the File Manager (FMS) and the Disk Utility Package (DUP). The resident disk handler is the only part of DOS in the O.S. ROM. FMS and DUP are located on diskette and are loaded ('booted') into the computer on power-up.

THE RESIDENT DISK HANDLER

The resident disk handler is the simplest part of DOS. The disk handler does not conform to the normal CIO calling sequence as does the other Device Handlers. The relationship of the Disk Handler to the I/O subsystem is shown in Figure 8.1 in section 8.2 of this book.

From figure 8.1 we can see that the DCB is the way to communicate to the Disk Handler. The calling sequence for the Disk Handler is:

```
                        ;caller has set up DCB
        JSR  DSKINV     ;system routine vector to the resident disk handler
        BPL  OKAY       ;Branch if success, Y Reg. = 1
                        ;Else y reg. = error status (DCBSTA also has error)
```

The disk handler supports five functions:

FORMAT	Issue a Format command to the Disk Controller.
READ SECTOR	Read a specified sector.
WRITE SECTOR	Write a specified sector.
WRITE/VERIFY SECTOR	Write sector, check sector to see if written.
STATUS	Ask the disk controller for its status.

The FORMAT command clears all the tracks on the diskette and the sector addresses are written on the tracks. No file structure is put on the diskette by this command.

The three sector I/O commands can be used to read and write sectors on the diskette. You can use them to implement your own file structure. Section 10 of the O.S. Manual has an example of using the disk handler to write a boot file.

The STATUS function is used to determine the status of the disk drive(s). You can use the Status command for several purposes. Since the timeout for a Status command is smaller then the other commands, you can use it to see if a specific disk drive is connected. If the Disk Handler returns a device timeout error, you know the disk is not connected.

DUP

DUP (Disk Utility Package) is a set of utilities for diskettes, familiarly seen as the DOS menu. DUP executes commands by calling FMS through CIO. The commands are:

A. DIRECTORY
B. RUN CARTRIDGE
C. COPY FILES
D. DELETE FILES
E. RENAME FILES
F. LOCK FILES
G. UNLOCK FILES
H. WRITE DOS FILES
I. FORMAT DISK
J. DUPLICATE DISK
K. SAVE BINARY FILE
L. LOAD BINARY FILE
M. RUN AT ADDRESS
N. WRITE MEM.SAV FILE
O. DUPLICATE FILE

FMS

FMS (File Management System) is a non-resident device handler that uses the normal device handler-CIO interface. FMS is not present in the O.S. ROM. It is booted in at power-up if a diskette containing DOS is present.

FMS, like the other device handlers, gets I/O control data from CIO. FMS then uses the resident disk handler to do I/O to the diskette. FMS is called by setting up an IOCB and calling CIO. FMS supports some special CIO functions not available to other handlers:

FORMAT FMS calls the resident Disk Handler to format the diskette. After a succesfull format, FMS writes some file structure data on the diskette.
NOTE FMS returns the current file pointer.
POINT FMS sets the file pointer to a specified value.

DISK I/O

You can access all the standard file I/O calls through CIO. In BASIC this means using the I/O commands, such as OPEN, CLOSE, GET, PUT. In assembly language you have to set up the IOCB yourself and call CIO. Let's use BASIC to provide an easy introduction to DOS.

To do any disk I/O, you must first OPEN a file. The BASIC syntax for the OPEN is:

```
OPEN #IOCB,ICAX1,0,"D:MYPROG.BAS"
```

The #IOCB selects one of the 7 IOCBs availaible from BASIC (BASIC itself uses IOCB #0). ICAX1 is the OPEN type code. The bits for the type code are:

BIT	7	6	5	5	3	2	1	0
	x	x	x	x	W	R	D	A

Where: A is Append
D is Directory
R is Read
W is Write
x is unused

The various values for ICAX1 are discussed in section 5 of the O.S. Manual. Some of the key things to note about the various OPEN modes are:

ICAX1=6 This is used to OPEN the diskette directory. Records READ are the diskette directory entries.
ICAX1=4 READ mode.
ICAX1=8 WRITE mode. Any file OPEN'ed in this mode is deleted. The first bytes written will be at the start of the file.
ICAX1=9 WRITE APPEND mode. The file is left intact. Bytes written to this file are put at the end of the file.
ICAX1=12 UPDATE mode. This mode allows both READ and WRITE to the file. Bytes READ/WRITE'en start at the first byte in the file.
ICAX1=13 Not supported.

Now that we know how to OPEN a file, let's see how to transfer data between our program and the disk. There are two types of I/O you can use, record or character.

Character I/O simply means that the data in a file is one list of bytes. DOS interprets this list of bytes as data (no imbedded control characters). This is an example of character data (all values hex): 00 23 4F 55 FF 34 21.

Record I/O means that data in a file is made up of a set of records. A record is a group of bytes delimited by an EOL ($9B). This is an example of two records:

```
00 23 4F 55 FF 34 9B 21 34 44 9B
 !  record 1       ! record 2  !
```

Record and character I/O to files can be done in any arbitrary order. As a matter of fact, data created as records can be read as characters. File data created as characters can be read as records. The only difference between character and record I/O is that records end with $9B. $9B is treated as ordinary data when using character I/O.

BASIC supports record I/O quite well. The commands PRINT and INPUT can be used to write and read records from files. BASIC does not support character I/O as well as it could. The commands GET and PUT allow you to read and write a single byte at a time.

The O.S. has the ability to read and write blocks of characters. This ability is not used by BASIC. Besides the length of the block, the O.S. allows you to specify the address of the block. To use the character block mode of the O.S. from BASIC, you can write an Assembly language module to be called from BASIC by the USR function. Figure 8.6 in section 8.2 of this book has an example of a subroutine to do character block I/O.

RANDOM ACCESS

One of the most important uses of the diskette is for random access of stored records in an arbitrary order. Using the I/O commands in conjunction with the special commands NOTE and POINT allows the creation and use of random access files.

NOTE and POINT read and update the file pointer respectively. DOS keeps a file pointer for each file currenty OPEN which tells the DOS the current location in the file. The file pointer has two parameters, the sector number and the byte count. The sector number (a value from 1-719) tells the DOS what sector on the diskette the file pointer is pointing to. The byte count is a count into the sector we are on (e.g. the first byte in a sector has a byte count of 0, the second byte is one, etc.). Figure 9.1 shows the relationship of the file pointer to the file. All values are hex. The file pointer values for the bytes in this file are given below the bytes in the file.

	A	B	C	EOL	D	E	F	EOL	G	H	I	J	k	EOL	A	B
File	41	42	43	9B	44	45	46	9B	47	48	49	4A	4B	9B...	41	42
File Pointer																
Sector Number	50	50	50	50	50	50	50	50	50	50	50	50	50	50...	50	51
Byte Count	0	1	2	3	4	5	6	7	8	9	A	B	C	D...	7C	0

The above file was created in BASIC by:

```
     10 OPEN #1,8,0,"D:FILE"
     20 ?#1;"ABC"
     30 ?#1;"DEF"
     40 ?#1;"GHIJ"
        :
        :           :REM Fill the rest of the sector
        :
     100 ?#1;"AB"    :REM This writes a record that crosses end of sector
     150 CLOSE #1
```

FIGURE 9.1 NOTE AND POINT VALUES

The sector number is 50 because DOS arbitrarily started this file on sector 50. The sector number changes to 51 because the file is longer than a sector. DOS linked the file to the next available sector, 51. The record "AB" crosses the end of the first sector.

The byte count of the file pointer starts at 0 and is incremented until the end of the sector, $7D (125 Dec.). DOS reserves the last 3 bytes of every sector for overhead data for the file. For files on the 810, the maximum byte count is 124 (0-124 = 125 total bytes). The maximun for the 815 is 252 (253 total bytes). When the file reaches the end of a sector, the Byte Count resets to 1.

You should now have a good idea how records are stored on the diskette and how to retreive them. Figure 9.2 is a subroutine to save records, keep track of where they are, and retrieve them. Appendix VIII is a complete random access method written in Basic.

```
1000 REM THIS ROUTINE CREATES AND ACCRESSES RANDOM ACCESS FILES FOR FIXED
LENGTH RECORDS
1001 REM COMMANDS ARE
1002 REM CMD=1 WRITE NTH RECORD
1003 REM CMD=2 READ NTH RECORD
1004 REM CMD=3 UPDATE NTH RECORD
1005 REM
1006 REM RECORD$IS THE INPUT/OUTPU RECORD
1007 REM N IS THE RECORD NUMBER
1010 REM INDEX IS A TWO DIMENSINAL ARRY DIM'ED INDEX(1,RECNUM). INDEX HOLDS
THE NOTE VALUES FOR ALL RECORDS
1020 REM IOCB1 IS THE ASSUMED DATA FILE
1100 REM
1200 ON CMD GOTO 2000,3000,4000
2000 REM
2100 REM WRITE NTH RECORD
2200 NOTE #1,X,Y
2300 INDEX(SEC,N)=X:INDEX(BYTE,N)=Y
2400 ? #1;RECORD$:RETURN
3000 REM
3010 REM READ NTH RECORD
3020 REM
3030 X=INDEX(SEC,N):Y=INDEX(BYTE,N)
3040 POINT #1,X,Y
3050 INPUT #1;RECORD$
3060 RETURN
4000 REM
4010 REM UPDATE NTH RECORD
4020 REM
4040 X=INDEX(SEC,N):Y=INDEX(BYTE,N)
4050 POINT #1,X,Y
4060 ? #1;RECORD$
4070 RETURN
```

FIGURE 9.2 NOTE AND POINT EXAMPLE

CHAPTER 10
SOUND

I. ATARI 400/800™ hardware capabilities

The ATARI 400/800™ has extensive sound capabilities. There are 4 sound channels, independently controllable. Each channel has a frequency register determining the note, and a control register regulating the volume and the noise content. Several options allow you to insert hi-pass filters, choose clock bases, set alternate modes of operation, and modify poly-counters.

For the purposes of this chapter, a few terms need to be clarified:

1 hz (hertz)	is 1 pulse per second
1 Khz (kilo-hertz)	is 1,000 pulses per second
1 Mhz (mega-hertz)	is 1,000,000 pulses per second

A pulse is henceforth defined to be a sudden voltage rise followed by a sudden voltage drop, internal to an electronic circuit, which, if sent to the TV speaker, will be turned into an audible pop (several in succession will be a buzz, etc.).

A "wave" as used here refers to a voltage change with respect to time, and also denotes a type of sound; i.e., waves created by the ATARI 400/800™ are square waves (like in figure 10.2), brass instruments produce triangle waves, and a singer produces sine waves (depicted in figure 10.15). If sent to the TV speaker, this voltage wave will be converted to a sound wave.

A single square wave is the same as a pulse as described above, and a continuous string of pulses is a square wave.

A shift register is like a memory byte (in that it holds binary data) that when instructed, shifts all its bits to the right one position; i.e., bit 5 will get whatever was in bit 4, bit 4 will get whatever was in bit 3, etc. Thus, the right-most bit is pushed out, and the left-most bit assumes the value on its input wire:

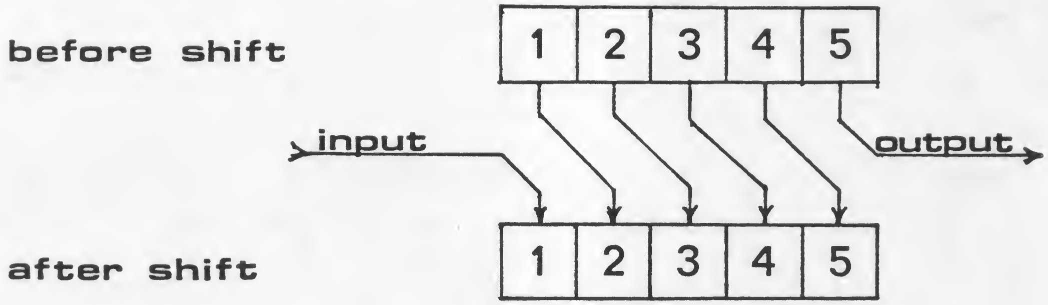


figure 10.1
diagram of bit flow of a shift register

AUDF1-4 is to be read, "any of the audio frequency registers, 1 through 4." Their addresses are: $D200, $D202, $D204, $D206 (53760, 53762, 53764, 53766).

AUDC1-4 is to be read, "any of the audio control registers, 1 through 4." Their addresses are: $D201, $D203, $D205, $D207 (53761, 53763, 53765, 53767).

The words "frequency", "note", "tone", and "pitch" are used interchangeably. Their meanings are identical for the purposes of this chapter.

"Noise" and "distortion" are used interchangeably although their meanings are not the same. "Noise" is a more accurate description of the function performed by the ATARI 400/800™.

The 60 hz interrupt referred to in part II of this chapter is also called the vertical blank interrupt.

All examples are in BASIC unless otherwise stated. Type the examples exactly as they appear; i.e., if there are no line numbers, don't use any, and if several statements are on the same line, type them as such.

You will encounter some problems when POKEing sounds in BASIC or in machine language. The POKEY chip needs to be initialized before it will work properly. To initialize in BASIC, do a null sound statement; i.e., SOUND 0,0,0,0. In machine language, store a 0 at AUDCTL ($D208 = 53768), and a 3 at SKCTL ($D20F = 53775).

AUDF1-4

Each audio channel has a corresponding frequency register which controls the note played by the computer.

The frequency register contains the number 'N' used in a divide by N circuit. This divide is not a division in the mathematical sense, but rather something much simpler: for every N pulses coming in, 1 pulse goes out. For example, below is a diagram of a divide by 4 function:

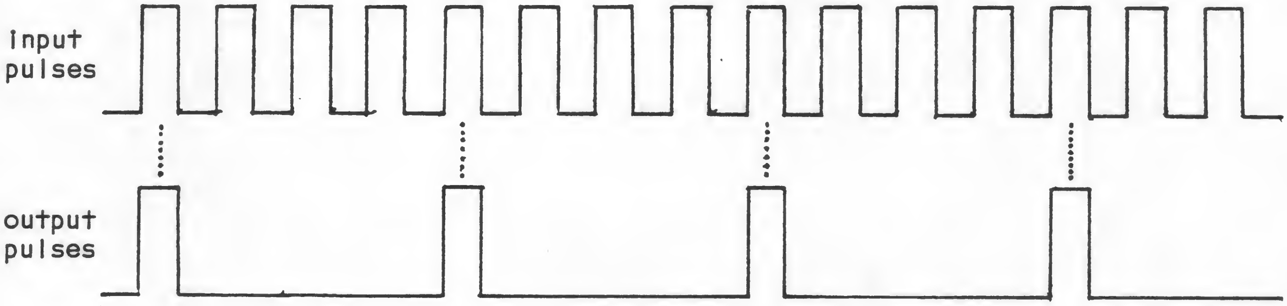


figure 10.2
divide by 4 diagram

As N gets larger, output pulses will become less frequent, making a lower note.

For the purposes of this discussion, frequency is a measure of the number of pulses in a given amount of time; i.e., a note with a frequency of 100 hz means that in one second, exactly 100 pulses will occur. The more frequent (hence the term "frequency") the pulses of a note, the higher the note. For example, a soprano singer sings at a high frequency (perhaps 5 Khz), and a cow moo's at a low frequency (perhaps 100 hz).

AUDC1-4

Each channel also has a corresponding control register. These registers allow the volume and distortion content of each channel to be set. The bit assignment for AUDC1-4 is as follows:

AUDC1-4

bit number	7	6	5	4	3	2	1	0
	distortion			vol only	volume			

figure 10.3
AUDC1-4 bit assignment

VOLUME: The volume control for each audio channel is very straightforward. The lower 4 bits of the audio control registers represent a 4 bit number that corresponds to a volume level. A zero in these bits means no volume, and a 15 means as loud as possible (there are 16 volume levels).

The sum of the volumes of the 4 channels should not exceed 32, since this forces overmodulation of the audio output. The sound produced tends to actually lose volume and assume a buzzing quality.

DISTORTION: Each channel also has 3 distortion control bits in its audio control register. Distortion is used to create special sound effects any time a pure tone is undesireable.

Atari's use of distortion is unique in the industry. Its versatility and controllability lend it well to the synthesis of an almost endless variety of sounds, from rumbles, rattles, and squawks to clicks, whispers, and mood setting background tempos from lurking ghosties or from outer space.

Distortion as defined by Atari is not equivalent to the standard interpretation. For example, "intermodulation distortion" and "harmonic distortion" are quality criteria specified for high fidelity stereo systems. These types of distortion refer to waveform degeneration, where the shape of the wave is slightly changed due to error in the electronic circuitry. Atari's distortion does not alter waves (they are always square waves), but rather deletes selected pulses from the waveform. This technique is not adequately characterized by the word "distortion". A more descriptive and appropriate term for Atari's distortion methods is "noise".

But before Atari's noise generation techniques can be fully grasped, you must understand poly counters.

Poly counters are employed in the ATARI 400/800™ as a source of random pulses used in noise generation. Atari's poly counters consist mostly of a shift register (see description of shift register at the beginning of the chapter) made to shift rapidly (1.79 Mhz). The shift register's output delivers the random pulses, and its input is determined by processing the values of selected shift register bits.

For example, in the diagram below, the old value of bit 5 will be pushed out of the shift register to become the next random pulse, and bit 1 will become a function of bits 3 and 5:

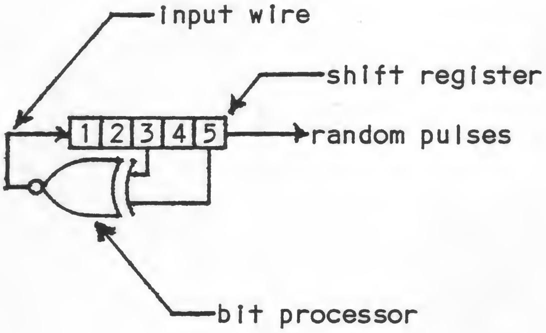


figure 10.4
5 bit poly-counter

The bit processor gets values from certain bits in the shift register (bits 3 and 5 above), and processes them in a way irrelevant to this discussion. It yields a value which is used to become bit #1 of the poly counter's shift register.

These poly counters are not truly random since they repeat their bit sequence after a certain span of time. As you might suspect, their repetition rate depends upon the length (the number of bits long) of the shift register used in the poly counter; i.e., the longer ones require many shifts before they repeat, while the shorter ones repeat more often.

With some background in poly counters, you are now equipped to understand how Atari generates distortion.

On the ATARI 400/800™, distortion is achieved by using random pulses from these poly counters in a selection circuit. This circuit is actually a digital comparator, but "selection circuit" is more descriptive and thus will used in this chapter.

The only pulses making it through the selection circuit to the output are ones coinciding with a random pulse. Various pulses from the input are thereby eliminated in a random fashion. The following illustrates this selection method. A dotted line connects pulses that coincided:

745-5236
tech assistance

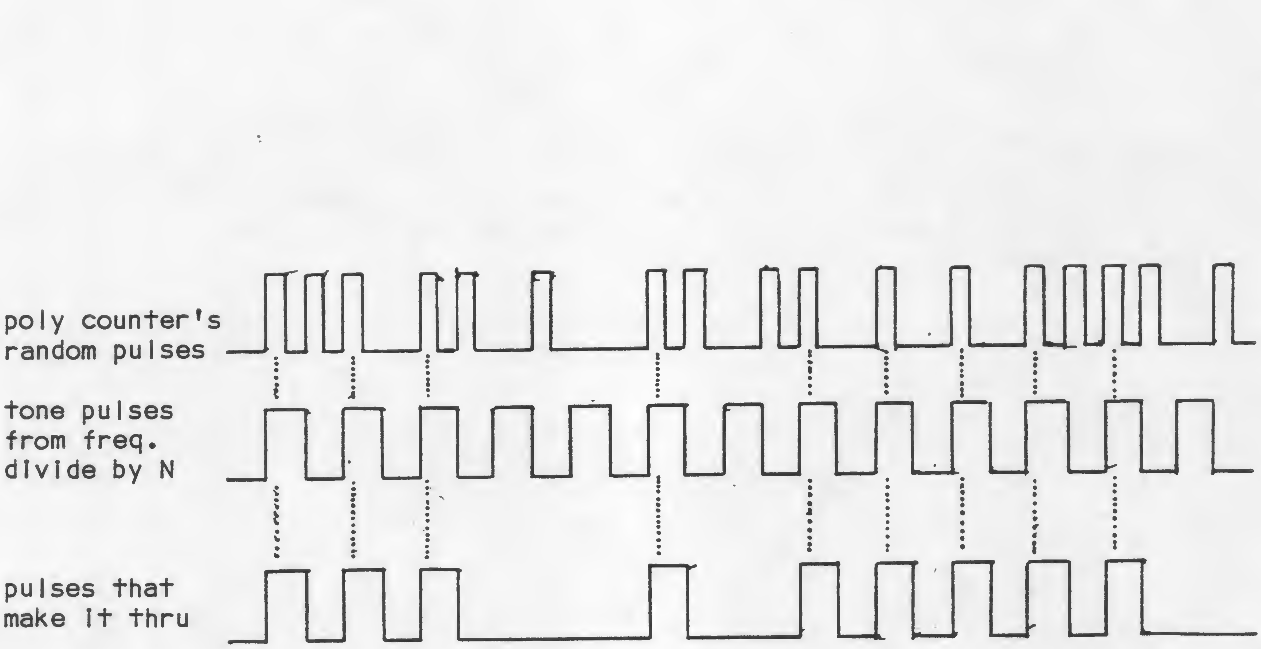

poly counter's random pulses
tone pulses from freq. divide by N
pulses that make it thru

figure 10.5
selection type function
used to mix in distortion

The net effect is this: some pulses from the frequency divide by circuit (discussed part I, section entitled AUDF1-4) are deleted. Obviously, if various pulses are deleted, the note will sound different. This is how distortion is introduced into a sound channel.

Since poly counters repeat their bit sequences, a pattern of pulses will be repeated. And since the selection circuit uses this pattern to delete pulses, the distorted note will be similarly patterned. It is this method which allows the programmer to create noises such as drones, motors, and other sounds having repetitive patterns.

The ATARI 400/800™ is equipped with three poly counters of different lengths, providing several levels of randomness. The smaller poly counters (4 and 5 bits long) repeat often enough to create droning sounds that rise and fall quickly, while the larger poly counter (17 bits long) takes so long to repeat that no pattern to the distortion is readily apparent. This 17 bit poly counter can be used for explosions, steam, and any sound where random crackling and popping is desired. It is even irregular enough to be used to generate white noise (an audio term meaning a hissing sound). For an example of white noise generation, try the following:

```
SOUND 0,100,8,8
POKE 53768,64
```

A 9 bit poly counter option offers a reasonable compromise between the short poly counters and the long one (see part I, discussion of AUDCTL).

Each audio channel offers 6 distinct combinations of the three poly counters:

AUDC1-4

7	6	5	4	3	2	1	0

7	6	5	
0	0	0	div clock by freq, select using 5 bit then 17 bit polys, div by 2
0	X	1	div clock by freq, select using 5 bit poly, then div by 2
0	1	0	div clock by freq, select using 5 bit then 4 bit polys, div by 2
1	0	0	div clock by freq, select using 17 bit poly, div by 2
1	X	1	div clock by freq, then div by 2 (no poly counters)
1	1	0	div clock by freq, select using 4 bit poly, div by 2

NOTES: "Clock" means the base frequency - see AUDCTL discussion, part I.

An 'X' means, "it doesn't matter if this bit is set or not." The structure of AUDC1-4 as shown in figure 10.7 will clarify why it doesn't matter.

figure 10.6
available poly counter combinations

These upper AUDC1-4 bits control three switches in the audio circuit as shown below. This diagram will help you understand why the table of figure 10.6 is structured as it is:

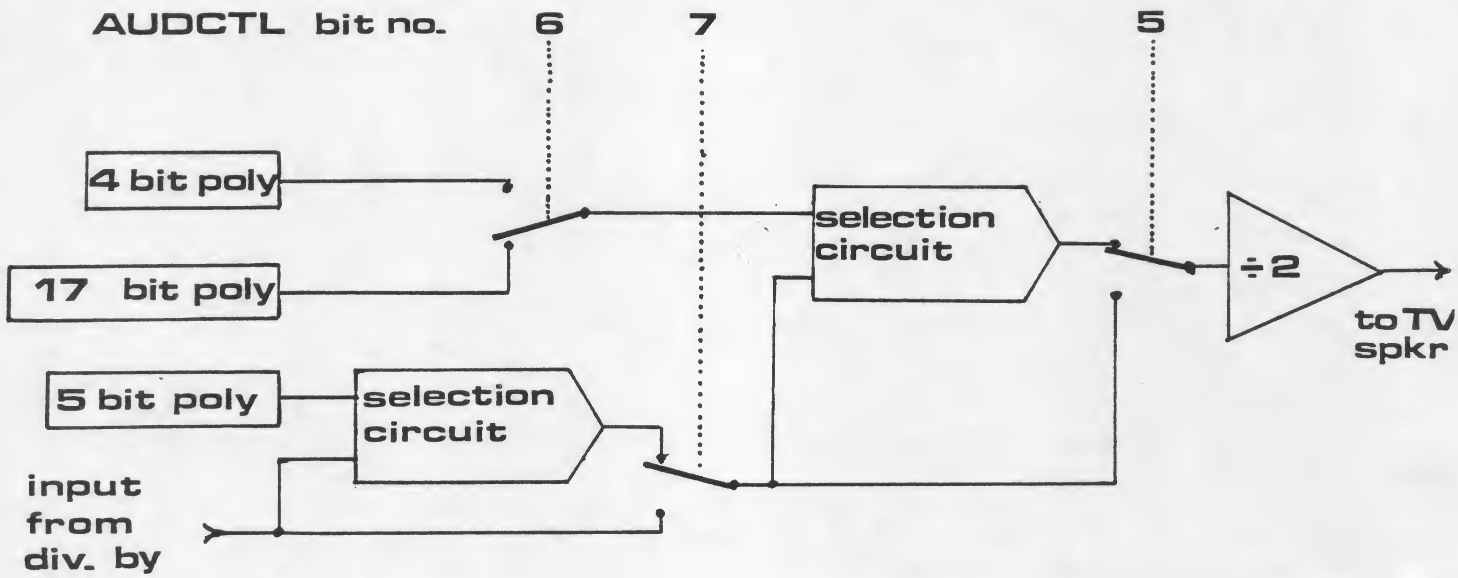


figure 10.7
AUDC1-4 block diagram

Each combination of the poly counters offers a unique sound. When striving for a particular sound, try each poly counter combination at several frequencies since one distortion setting can sound completely different at different frequencies. Below is a table of suppositions, just to get you started:

AUDC1-4

7 6 5 4 3 2 1 0

7 6 5	low frequencies middle frequencies high frequencies
0 0 0	geiger counter raging fire rushing air steam
0 X 1	machine gun auto at idle electric motor power transformer
0 1 0	calm fire laboring auto auto with a "miss"
1 0 0	building crashing in radio interference waterfall
1 X 1	pure tones
1 1 0	airplane lawn mower electric razor

figure 10.8
sounds produced by distortion combinations
at several frequencies

Summary of distortion: A shift register is used as the main component of a poly counter, which is used to generate random pulses. The random pulses are used to delete selected note pulses, thereby introducing distortion into a channel. The upper bits (bits 7,6,5) of AUDC1-4 toggle 3 switches which select the poly counters to be used in that channel for distortion.

VOLUME ONLY: Bit #4 of AUDC1-4 specifies a volume only mode. When this bit is set, the volume bits (AUDC1-4 bits 0-3) are sent to the TV as volume; i.e., no frequency is associated with this bit.

To fully understand the use of this mode of operation, you must understand how a speaker works and what happens to the TV speaker when it receives a pulse.

Any speaker has a cone which moves in and out. The cone's position at any time is directly proportional to the voltage it is receiving from the computer at that time; i.e., if the voltage sent is zero, then the speaker is in the resting position, and if the voltage sent is 5, then the speaker is in the extended position. And whenever the cone changes position, it moves air which is detected by your ear as sound.

From our definition of a pulse, you know that it consists of a rising voltage followed by a falling voltage. If we were to send the speaker a pulse, it would push out with the rising voltage, and pull back with the falling voltage, resulting in a wave of air which can be detected by your ear as a pop. The following statements will produce such a pop on the TV speaker by sending a single pulse:

```
POKE 53761,31:POKE 53761,16
```

A stream of pulses (or wave) would set the speaker into constant motion, and a continuous buzz would be heard. This is how the computer generates sound on the TV speaker.

It is essential to note that the volume sent does not drop back to zero, but rather remains until the program changes it. The program is expected to modulate the volume often enough to create a noise. Now try the following, listening carefully after each statement:

```
POKE 53761,31
POKE 53761,31
```

The first time you heard a pop, which is as expected; i.e., the speaker pushed out and moved air. But the second time you didn't. This is because the speaker cone is already in the extended position - another extension command does nothing to the speaker, moving no air, so you hear nothing. Now try this:

```
POKE 53761,16
POKE 53761,16
```

Just like before, you heard a pop the first time as the speaker moved back to its resting position, and you heard nothing the second time since the speaker was already in the resting position.

Thus, the volume only bit allows the program complete control over the position of the speaker at any time. Although the examples given above are only binary examples (either all on or all off), you are by no means limited to this type of speaker modulation. You are actually allowed to force the speaker to any of 16 distinct positions.

For example, a simple triangle wave (similar to the waveform produced by brass instruments) could be generated by sending a volume of 8 followed by 9, 10, 11, 10, 9, 8, 7, 6, 5, 6, 7, and back to 8, and repeating this sequence over and over very rapidly (BASIC is too slow). In this example, 7 of the possible 16 speaker positions are used. (The sample program given under the machine code sound generation section would in fact make a triangle wave if its duration table contained all 1's).

Therefore, by changing the volume quickly virtually any waveform can be created. It is feasible, for example, to make the computer say, "Hello" in a clear voice.

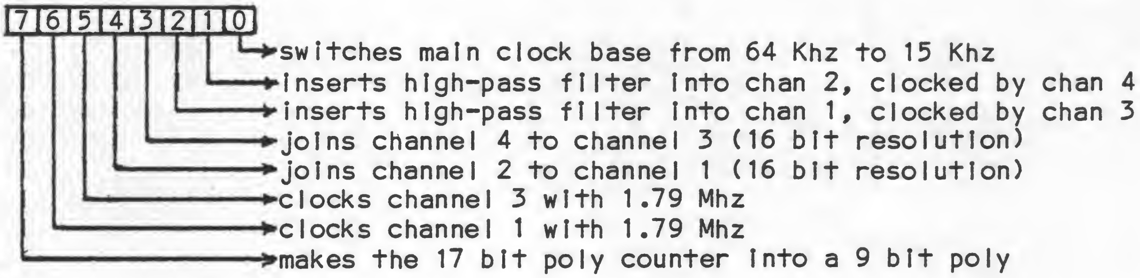
If you have never been exposed to wave theory, audio, or electronics, the purpose and use of the volume only bit will seem very convoluted. It would probably be best in this case to buy a book on the subject.

There is a continued discussion of this bit in part II.

AUDCTL

In addition to the independent channel control bytes (AUDC1-4), there is an option byte (AUDCTL) affecting all the channels. Each bit in AUDCTL is assigned a specific function:

AUDCTL ($D208 = 53768)   if set, this bit...



switches main clock base from 64 Khz to 15 Khz
inserts high-pass filter into chan 2, clocked by chan 4
inserts high-pass filter into chan 1, clocked by chan 3
joins channel 4 to channel 3 (16 bit resolution)
joins channel 2 to channel 1 (16 bit resolution)
clocks channel 3 with 1.79 Mhz
clocks channel 1 with 1.79 Mhz
makes the 17 bit poly counter into a 9 bit poly

figure 10.9
AUDCTL bit assignment

CLOCKING: Before proceeding with the explanations of the AUDCTL options, clocking must be well understood. In general, a clock is a train of pulses used to synchronize the millions of minute internal operations occurring every second in any computer. A clock pulses continuously, each pulse telling the circuitry to perform another step in its operation.

The section on frequency explained that a frequency divider outputs only every Nth input pulse. But where do the input pulses come from? The clock!

Several clocks are used in the computer, and AUDCTL allows you to change the clock used as the input to the divide by N to either a faster or a slower clock. When this input clock changes, the output from the frequency divide by changes proportionally.

For example, imagine the clock to be pulsing at a rate of 15 Khz, and the frequency register is set to divide by 5. The rate of output pulses from the divide by circuit would be 3 Khz. But if we changed the clock, or input frequency, to 40 Khz without changing the frequency register, then what would happen? The divide by N would still be outputting every 5th pulse, but it is receiving input at a faster rate, and thus every 5th pulse comes along sooner. The result is an output frequency (from the divide by N) of 8 Khz.

The formula for the output frequency (from the divide by N) is quite simple:

$$\text{output frequency} = \frac{\text{clock}}{\text{N}}$$

This shows that the output frequency is directly proportional to the input clock.

So, if someone were to make the clock speed up, then everything using that clock would execute with less delay. In the case of the ATARI 400/800™ audio circuitry, if the clock rate was increased, then every sound pulse to the TV would be packed closer together, making a higher frequency, or tone (see part I, section on frequency).

The inverse is true as well - if the clock is slowed, all pulses would spread out, lowering the frequency.

15 Khz OPTION: Try the following:

```
SOUND 0,128,10,8                               medium tone
POKE 53768,1                                   AUDCTL 15 Khz option
```

As explained in the section on clocking, a slower clock produces a less frequent string of output pulses, or a lower sound. Location 53768 is AUDCTL, and a 1 sets bit #0, which switches the 64 Khz clock into a 15 Khz clock. In the above demonstration, you heard the frequency drop to about 1/4th the original frequency, since the clock dropped to 1/4th its original rate.

It is important to note that if this bit is set, every sound channel clocked off 64 Khz will be changed to the 15 Khz clock base.

1.79 Mhz OPTIONS: Try the following:

```
SOUND 0,255,10,8                             turn on channel #1, low tone
POKE 53768,64                                set AUDCTL bit #6
```

As explained in section on clocking, changing the time base proportionally changes the frequency heard. In this case, putting a 64 into AUDCTL causes channel #1 to use the 1.79 Mhz time base rather than the 64 Khz time base. As you might have expected from the previous section, the POKE caused the tone to become higher. The change made a substantial difference since 1.79 Mhz is almost 30 times faster than 64 Khz.

Setting AUDCTL bit #5 forces channel #3 to use 1.79 Mhz as its clock in the same manner:

```
SOUND 2,255,10,8                            turn on channel #3, low tone
POKE 53768,32                               set AUDCTL bit #5
```

These options extend the range of the frequencies capable of being generated to well beyond the limits of the human ear (we can hear up to about 20 Khz).

16 BIT OPTIONS: AUDCTL bits 3 and 4 allow two channels to be joined, creating a single channel with an extended dynamic range. Working independently, each channel's frequency can range from 0 to 255 (8 bits of divide by N capability). Joining two channels allows a frequency range of 0 to 65535 (16 bits of divide by N capability). In this mode it is possible to reduce the output frequency to single pops separated by several seconds. The following program uses two channels in the 16 bit mode, and two paddles as the frequency inputs. Insert a set of paddles into port #1, type in and run the following program:

```
10 SOUND 0,0,0,0                      initialize sound
20 POKE 53768,80                      clock ch1 w 1.79 Mhz, clock ch2 w ch1
30 POKE 53761,160:POKE 53763,168      turn off ch1, turn on ch2 (pure tones)
40 POKE 53760,PADDLE(0):POKE 53762,PADDLE(1)
50 GOTO 40                            use paddles to put freqs in freq regs
```

The right paddle tunes the sound coarsely, and the left paddle finely tunes the sound between the coarse increments.

This program first sets bits 4 and 6 of AUDCTL which means, "clock channel #1 with 1.79 Mhz, and join channel #2 to channel #1." Once this happens, the 8 bit frequency registers of both channels are assumed to represent a single 16 bit number N, used to divide the input clock.

Next, channel #1's volume is set to zero. This is because the output from channel #1 is meaningless to anything but channel #2. When these channels are put into their 16 bit mode, internal changes cause the two channels to act as one 16 bit divide by N. These changes invalidate the output of the first channel.

Channel #1's frequency register is used as the fine or low byte in the sound generation, and channel #2's frequency register is the coarse or high byte. For example, POKEing a 1 into channel #1's frequency register makes the pair divide by 1. POKEing a 1 into channel #2's frequency register makes the pair divide by 256. And POKEing a 1 into both frequency registers makes the pair divide by 257.

Bit #3 of AUDCTL can be used to join channel #4 to channel #3 in precisely the same way.

The 16 bit option is useful in applications where a finer control over the frequency is desired. These finer increments are achieved by making the input clock 1.79 Mhz, and using one of the 16 bit options as in the above program, line 20.

To understand why the increments are finer, you must understand what happens to the output frequency as the value in the frequency register changes. First a concrete example: if the frequency register is dividing by one, and the input frequency is 1 Khz, the output is 1 Khz. Now increment the frequency register so that it is dividing by 2, and we get an output of 500 hz, which is a drop of 500 hz. Increment the frequency register once more so that it is divides by 3, and the output becomes 333 hz, a drop of 267 hz. The second increment had less effect than the first (the first increment affected a change of 500 hz, while the second increment affected only a 267 hz change). If we were to increment further, each step would continue to have less effect than its predecessor.

The idea is basically this: the larger the number N in the frequency divide by register, the smaller the difference in output frequency as N changes.

Therefore, while clocking the 16 bit channel with 1.79 Mhz then dividing by several thousand will yield the same output frequency as an 8 bit channel clocked with 64 Khz and dividing by several hundred, the 16 bit channel will offer finer resolution since each increment (or decrement) causes less effect.

For a little 16 bit variety try the following statements, and experiment with various value combinations in the last 4 POKEs:

```
SOUND 0,0,0,0
POKE 53768,24      D208< 18  AUDCTL
POKE 53761,168     D201< A8 }
POKE 53763,168     D203<    }  AUDCX NOISE, VOL
POKE 53765,168     D205<    }
POKE 53767,168     D207<    }    D200< F0
POKE 53760,240:REM try POKEing other numbers into these next 4 locations
POKE 53764,252     D204 < FC }  AUDF  FREQ
POKE 53762,28      D202 < 1C }
POKE 53766,49      D206 < 31 }
                   weird sound!
```

HIGH-PASS FILTERS: AUDCTL bits 1 and 2 control high pass filters in channels 2 and 1 respectively. A high-pass filter is indeed a filter in that it allows only selected things to pass through. As the name implies, the things allowed to pass are high frequencies.

In the case of these high-pass filters, high frequencies are defined to be anything higher than the clock. The clock in this case is the output of some other channel.

For example, if channel #3 is playing a cow's moo, and AUDCTL bit #2 is set, then only a noise or sound with a frequency higher than the moo will be heard on channel #1 (anything lower than the "moooo" will be filtered out):

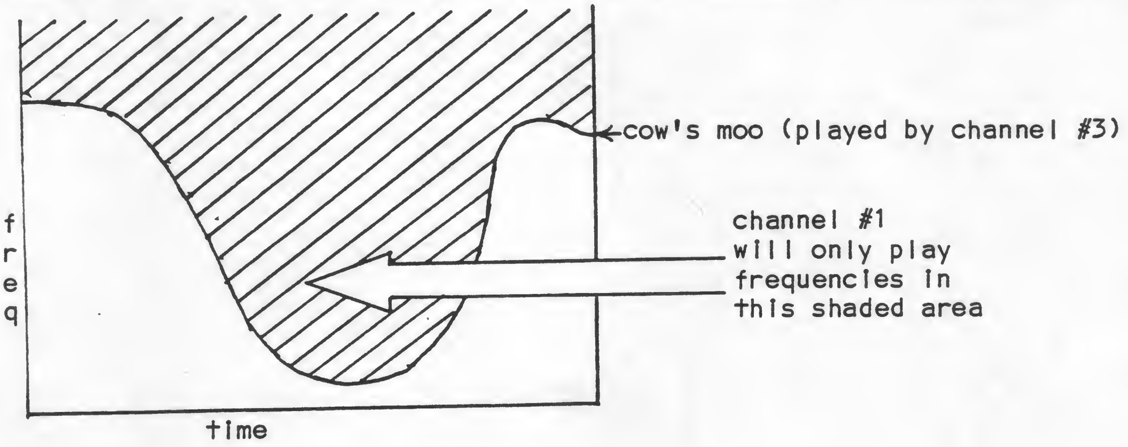

cow's moo (played by channel #3)
channel #1 will only play frequencies in this shaded area
freq
time

figure 10.10
diagram of the effect of a high-pass filter
inserted in channel #1 and clocked by channel #3

The filter is programmable in real time since the clock used is actually another channel which can be changed on the fly. This opens a large field of possibilities to the programmer.

The filters are used mostly to create special effects. Try the following:

```
                A8
                              A8
SOUND 0,0,0,0       D208<4
POKE 53768,4
POKE 53761,168:POKE 53765,168
POKE 53760,254:POKE 53764,127      musical  siren

                FD            7F
```

9 BIT POLY CONVERSION: Bit #7 of AUDCTL turns the 17 bit poly counter into a 9 bit poly counter. The discussion of poly counters explained that the shorter the poly counter, the more often its distortion repeats, or the more discernable the pattern in the distortion. Therefore, it is reasonable to assume that changing the 17 bit poly into a 9 bit poly will make the noise pattern more discernable.

Try the following demonstration of the 9 bit poly option, listening carefully when the POKE is executed:

```
SOUND 0,80,8,8                          use the 17 bit poly
POKE 53768,128                          change to the 9 bit poly
```

II. Sound Generation Software Techniques

There are two basic ways to use the ATARI 400/800™ sound system: static and dynamic. Static means that the program sets a few sound generators, waits, then turns them off. Dynamic means that the sound generators are continuously updated as the program is executing. For example:

```
 static sound                                           dynamic sound

SOUND 0,120,8,8                                       FOR X=0 TO 255
                                                        SOUND 0,X,8,8
                                                      NEXT X
```

Static sound

Static sound is quite limited. Mostly, the only sounds that can be made are beeps, clicks, and buzzes. However, there are a few exceptions to this rule. Two examples are the programs given as special effects in part I, sections on high-pass filters, and 16 bit sound. Another simpler example is using two sound channels in the following way:

```
SOUND 0,255,10,8
SOUND 1,254,10,8
```

slow shift

The strange effect is a result of closely phased peaks and valleys. Examine the diagram below. It shows two channels independently running sine waves at slightly different frequencies, and their sum. The sum curve shows the strange interference pattern created when these two channels are added.

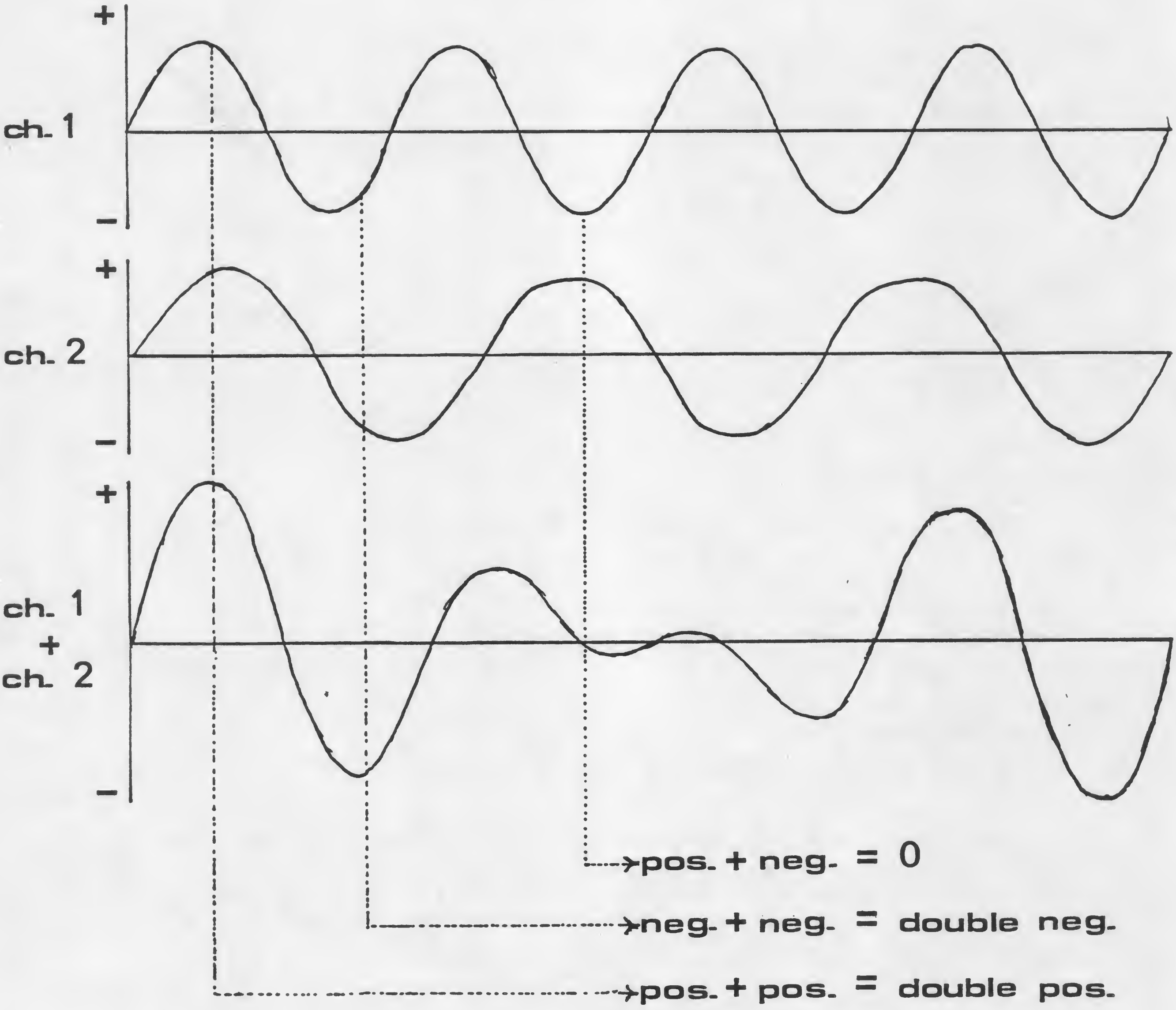


figure 10.11
two sine waves at different frequencies,
and their sum

Before the sound channels are sent to the TV they are mixed together. The mixing process is similar to the mixing performed by the human ear; a simple addition.

Figure 10.11 shows that at some points in time the waves are assisting each other, and at other points, they are in direct conflict. Adding the volumes of two waves whose peaks have coincided will yield a wave with twice the strength, or volume. Likewise adding the volumes of two waves while one is peaking and the other is at a low will result in a cancellation of both of them.

On the graph of the sum curve, we can see this in action. Toward the ends of the graph, volume increases since both channels' peaks and valleys are close together, almost doubling the sound. And toward the middle of the graph the waves oppose each other and the resulting wave is flat. An interesting project might be writing a program to plot interaction patterns of 2, 3, and 4 channels like in figure 10.11. You could possibly discover some very unique sounds.

The more slight the difference in frequency between the two channels, the longer the pattern. To understand this, draw yourself some graphs similar to figure 10.11 and study the interaction. As an example, try the following statements:

```
SOUND 0,255,10,8
SOUND 1,254,10,8
SOUND 1,253,10,8
SOUND 1,252,10,8
```

There are many other similar examples, using the same principle:

```
SOUND 0,254,10,8
SOUND 1,127,10,8
```

Dynamic sound

In general, any sound that is not a simple beep, click or buzz must be generated using dynamic sound. Three levels of dynamic sound are available to the ATARI 400/800™ programmer: sound in BASIC, 60 hz interrupt, and sound in machine code.

BASIC SOUND: BASIC is somewhat limited in its sound capabilities. As you may have noticed, any SOUND statement kills any modified AUDCTL setting.

This potential problem can be avoided by POKEing in sounds rather than using the SOUND statement. The sample program given in part I under AUDCTL's 16 bit options is a good example of such a technique.

In addition, BASIC is limitted on account of its speed. If the program is not entirely dedicated to sound generation, there is generally not enough processor time to do more than static sound or choppy dynamic sound. The alternative is to temporarily halt all other processing while generating sound.

Another problem can occur when using the computer to play music on more than one channel. If all 4 channels are used, the time separation between the first sound statement and the fourth can be substantial enough to make a noticeably obnoxious delay between the melody and the harmony.

The following is a solution to this dilemma:

```
10 SOUND 0,0,0,0:DIM SIMUL$(16)
20 RESTORE 9999:X=1
25 READ Q:IF Q<>-1 THEN SIMUL$(X)=CHR$(Q):X=X+1:GOTO 25
27 RESTORE 100
30 READ F1,C1,F2,C2,F3,C3,F4,C4
40 IF F1=-1 THEN END
50 X=USR(ADR(SIMUL$),F1,C1,F2,C2,F3,C3,F4,C4)
55 FOR X=0 TO 150:NEXT X
60 GOTO 30
100 DATA 182,168,0,0,0,0,0,0
110 DATA 162,168,182,166,0,0,0,0
120 DATA 144,168,162,166,35,166,0,0
130 DATA 128,168,144,166,40,166,35,166
140 DATA 121,168,128,166,45,166,40,166
150 DATA 108,168,121,166,47,166,45,166
160 DATA 96,168,108,166,53,166,47,166
170 DATA 91,168,96,166,60,166,53,166
999 DATA -1,0,0,0,0,0,0,0
9000 REM
9010 REM
9020 REM this data contains the machine lang. program,
9030 REM and is read into SIMUL$
9999 DATA 104,133,203,162,0,104,104,157,0,210,232,228,203,208,246,96,-1
```

In this program, SIMUL$ is a tiny machine language program that "pokes" all four sound channels virtually simultaneously. Any program using SIMUL$ can accurately phase all four channels.

Any program can call SIMUL$ by simply putting the sound register values inside the USR function as shown above in line 50. The parameters should be ordered as shown, with the control register value following the frequency register value, and repeating this ordering 1 to 4 times, once for each sound channel to be set.

As a speed consideration as well as a convenience, SIMUL$ allows you to specify sound for less than 4 channels; i.e., 1, 2, and 3 or 1 and 2, or just channel 1. Simply don't put the unused parameters inside the USR function. The following shows a SIMUL$ call with only the first 2 channels:

```
X=USR(ADR(SIMUL$),F1,C1,F2,C2)
```

SIMUL$ offers another distinct advantage to the BASIC programmer. As mentioned earlier, the AUDCTL register is reset upon execution of any SOUND statement in BASIC. However, using SIMUL$, no SOUND statements are executed, and thus the AUDCTL setting is retained.

There is one other very impractical method of sound generation in BASIC. This is using the volume only bit of any of the four audio control registers. Type in and run the following:

```
SOUND 0,0,0,0
10 POKE 53761,16:POKE 53761,31:GOTO 10
```

This program sets the volume only bit in channel #1 and modulates the volume from 0 to 15 as fast as BASIC can. Practically speaking, this program uses 100% of the processing time available, and produces only a low buzz.

The best way to get complex sounds in BASIC without sacrificing the entire processor is using the 60 hz interrupt technique described in the next section.

60 HZ INTERRUPT: This technique is probably the most versatile and practical of all methods available to the ATARI 400/800™ programmer.

Precisely every 60th of a second, the computer hardware automatically generates what is termed an "interrupt". When this happens, the computer temporarily leaves the mainline program, (the program running on the system; i.e., BASIC, STAR RAIDERS™). It then executes an interrupt service routine, a very small routine designed specifically for servicing these interrupts. When the interrupt service routine finishes, it executes a special machine language instruction which restores the computer to the interrupted program. This all occurs in such a way (if done properly) that the program executing is not affected, and in fact has no idea that it ever stopped!

The interrupt service routine currently resident on the ATARI 400/800™ simply maintains timers, translates controller information and miscellaneous other chores requiring regular attention.

But before the interrupt service routine returns to the mainline program, it can be made to execute any user routine; i.e., your sound generation routine. This is ideal for sound generation since the timing is precisely controlled, and especially since another program can be executing without paying heed to the sound generator!

Even more impressive is its versatility. An interrupt sound program will lend itself equally well to a mainline program written in any language - BASIC, assembler, FORTH, PASCAL. In fact, the sound generator will require few if any modifications to work with another program, or even another language.

A table-driven routine offers maximum flexibility and simplicity for such a purpose. "Table-driven" refers to a type of program which accesses data tables in memory for its information. In the case of the sound generator, the data tables would contain the frequency values and possibly the audio control register values. The routine would simply read the next entries in the data table, and put them into their respective audio registers. Using this method, notes could change as often as 60 times per second, fast enough for most applications.

This program must be written in machine language since it is actually becoming a part of the operating system.

Once such a program has been written and placed in memory (say, at location $600), you need to install it as a part of the 60 hz interrupt service routine. This is accomplished by a method known as vector stealing, and is described further in appendix I.

Memory locations $224,$225 contain the address of a small routine called XITVBL (eXIT Vertical BLank interrupt service routine). XITVBL is designed to be executed after all 60 hz interrupt processing is complete, restoring the computer to the mainline program as previously discussed.

The procedure to install your sound routine is as follows:

1) place your program in memory
2) verify that the last instruction executed is a JMP $E462.
   ($E462 is XITVBL, so this will make the mainline program continue)
3) load the x register with the high byte of your routine's address.
   (a 6 in this case)
4) load the y register with the low byte of your routine's address.
   (a 0 in this case)
5) load the accumulator with a 7.
6) do a JSR $E45C (to set locations $224,$225)

Steps 3-6 above are all required to change the value of $224,$225 without error. The routine called is SETVBV (SET Vertical Blank Vectors), which will simply put the address of your routine into locations $224,$225 (see appendix I).

Once installed, the system will work as follows when an interrupt occurs:

1) the computer's interrupt routine is executed
2) it jumps to the program whose address is in $224,$225, which is now your routine.
3) your routine executes.
4) your routine then jumps to XITVBL.
5) XITVBL restores the computer and makes it resume normal operation.

For comparison, the following shows the interrupt sequence without your routine as a part:

1) the computer's interrupt routine executes.
2) it jumps to the address specified in $224,225, which is XITVBL.
3) XITVBL restores the computer to its pre-interrupt state, and makes the computer resume its mainline processing.

If you do not wish to implement such a program yourself, there is one available. A BASIC editor allows creation and modification of sound data while you listen. It is accompanied by an interrupt sound generator as described above: table driven, compatible with any language. The package is called INSOMNIA (INterrupt SOuNd Initializer/Alterer), and is offered by the Atari Program Exchange.

MACHINE CODE SOUND GENERATION: Using assembly language opens new doors in sound generation. An attempt can now be made to simulate particular musical instruments. The technique is as follows: write a program similar to the 60 hz interrupt routine in that it is table-driven. The output of that routine will look something like this for 3 music notes:

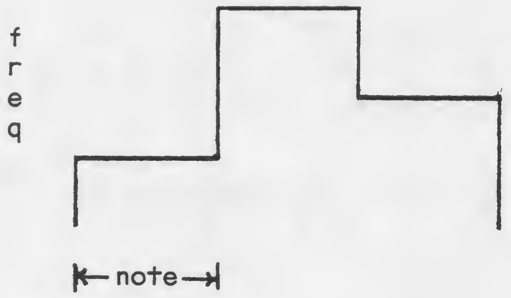


figure 10.12
example of 3 music notes played on a standard music routine

Since much more processing time is available, we can go a level deeper, minutely changing the frequency during the note's playing time so that it simulates an instrument. For example, suppose we discovered that whenever a piano key is struck (any key) we can get an identical sound by very quickly playing a table of frequencies. That table may look something like this:

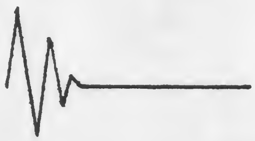


figure 10.13
graphed table of frequencies that possibly duplicates a piano key

Let's call the above table the "envelope table", and its data a "piano envelope". To simulate a piano, the idea would be to very quickly add the piano envelope to the plain vanilla beep. The note is thus slightly modified during its playing time. For example, a piano simulation of the 3 notes in figure 10.12 would look like this:

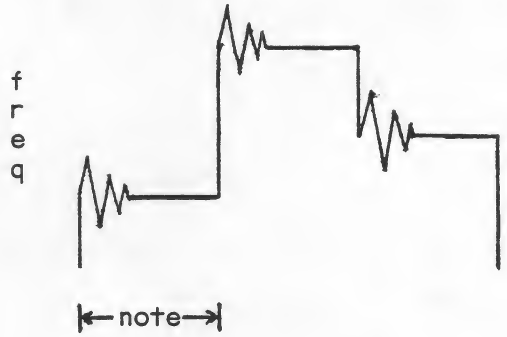


figure 10.14
example of the 3 notes of fig 10.10
played with a piano envelope

We have essentially the same sound produced by the standard music routine of figure 10.12, only the notes now have a piano flavor, and sound much prettier than just the flat "beeps".

Unfortunately, we had to sacrifice all other processing to get that piano flavor. The sound is no longer updated only once every note, but perhaps 100 times within the note's duration.

Volume only: Earlier we experimented with the AUDC1-4 volume only bits, and hinted at a lurking power, but discovered that they seemingly weren't of much use. This was due entirely to the fact that BASIC is too slow to effectively use them. Not so with machine language.

As mentioned earlier, this bit offers a tremendous capacity for accurate sound reproduction. True waveform generation (to the time and volume resolution limits of the computer) is now possible. Instead of just putting a piano flavor into the music, you can now make it duplicate a piano sound.

Unfortunately, it can never precisely simulate an instrument. 4 bits (16 values) is not enough volume resolution, although by no means is this technique rendered fruitless. The following program shows the use of one of the volume only bits. If you have an assembler, type it in and try it:

```
0100 ;
0110 ; VONLY          Bob Fraser 7-23-81
0120 ;
0130 ;
014u ; volume-only AUDC1-4 bit test routine
0150 ;
0160 ;
0170 ;
0180 ;
0190 AUDCTL=$D208
0200 AUDF1=$D200
0210 AUDC1=$D201
0220 SKCTL=$D20F
023u ;
0240 ;
0250  *=$B0
0260 TEMPO .BYTE 1
0270 MSC .BYTE 0
0280 ;
0290 ;
0300 ;
0310  *=$4000
0320  LDA #0
0330  STA AUDCTL
0340  LDA #3
0350  STA SKCTL
0360  LDX #0
0370 ;
0380  LDA #0
0390  STA $D40E kill vbi's
0400  STA $D20E kill irq's
0410  STA $D400 kill dma
0420 ;
0430 ;
0440 ;
0450 LOO LDA DTAB,X
0460  STA MSC
0470 ;
0480  LDA VTAB,X
0490 LO LDY TEMPO
0500  STA AUDC1
0510 L1 DEY
0520  BNE L1
0530 ;
054u ; dec most sig ctr
0550  DEC MSC
056u  BNE LO
0570 ;
0580 ;
0590 ; new note
0600 ;
0610  INX
0620  CPX NC
063u  BNE LOO
0640 ;
```

```
0650 ; wrap note pointer
0660  LDX #0
0670  BEQ LOO
0680 ;
0690 ;
0700 NC .BYTE 28 note count
0710 ;
0720 ; table of volumes to be played in succession
0730 VTAB
0740  .BYTE 24,25,26,27,28,29,30,31
0750  .BYTE 30,29,28,27,26,25,24
0760  .BYTE 23,22,21,20,19,18,17
0770  .BYTE 18,19,20,21,22,23
0780 ;
0790 ; this table contains the duration of each entry above
0800 DTAB
0810  .BYTE 1,1,1,2,2,2,3,6
0820  .BYTE 3,2,2,2,1,1,1
0830  .BYTE 1,1,2,2,2,3,6
0840  .BYTE 3,2,2,2,1,1
```

constant pitch
tone
(no big deal)

Suprisingly, speed is not really problem here. The wave has almost 60 steps, and the program can still be made to play the wave at a piercing level (approx. 10 Khz).

Remove lines 400-410, and try the program once more. It will sound quite broken up. The cause is the 60 hz interrupt discussed in the previous section. You can actually hear the interrupts taking place since all sound stops during that time.

Line 420 disables screen DMA. This is why the screen goes to solid background color when the program is executed. It serves two purposes: to speed up the processor, and to make the timing consistent, since DMA steals cycles at odd intervals. See Chapter 5 for more on DMA.

In this particular case, the sound created is a sine wave. The wave is remarkably pure, and does indeed sound like a sine wave. If graphed, the data looks like this:

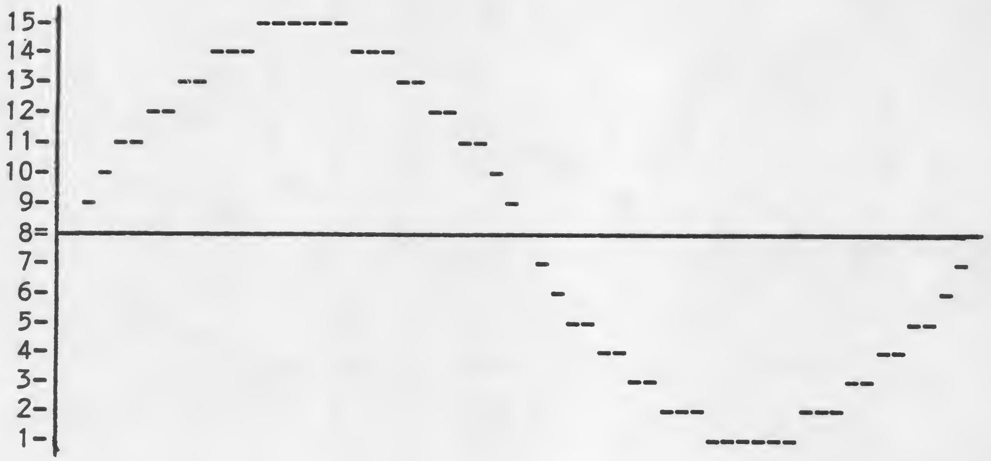


figure 10.15
graphed sine wave data for volume only program (above)

Much can be done with the ATARI 400/800™'s sound capabilities. The question is, "Why sound?"

Movie makers have long understood the importance of mood setting backgroud music. The recent space adventure movies by George Lucas are excellent examples. When the villian enters the room you know immediately to fear and hate him from the menacing background rhythms accompanying his entry. And you gleefully clap your hands when the hero saves the princess while gallant music plays excitedly in the background. Horror films can frighten you by just playing eerie music, even though the action may be completely ordinary.

SPACE INVADERS™ issues a personal threat to its player and victim with its echoing stomp. As the tempo increases, knuckles whiten and teeth grind. When a Zylon from STAR RAIDERS™ fires a photon torpedo you push frantically on the control to avoid impact. As it bores straight for your forehead, time slows and you hear it hissing louder and louder as it approaches. Just before impact, you duck and dislodge yourself from your armchair.

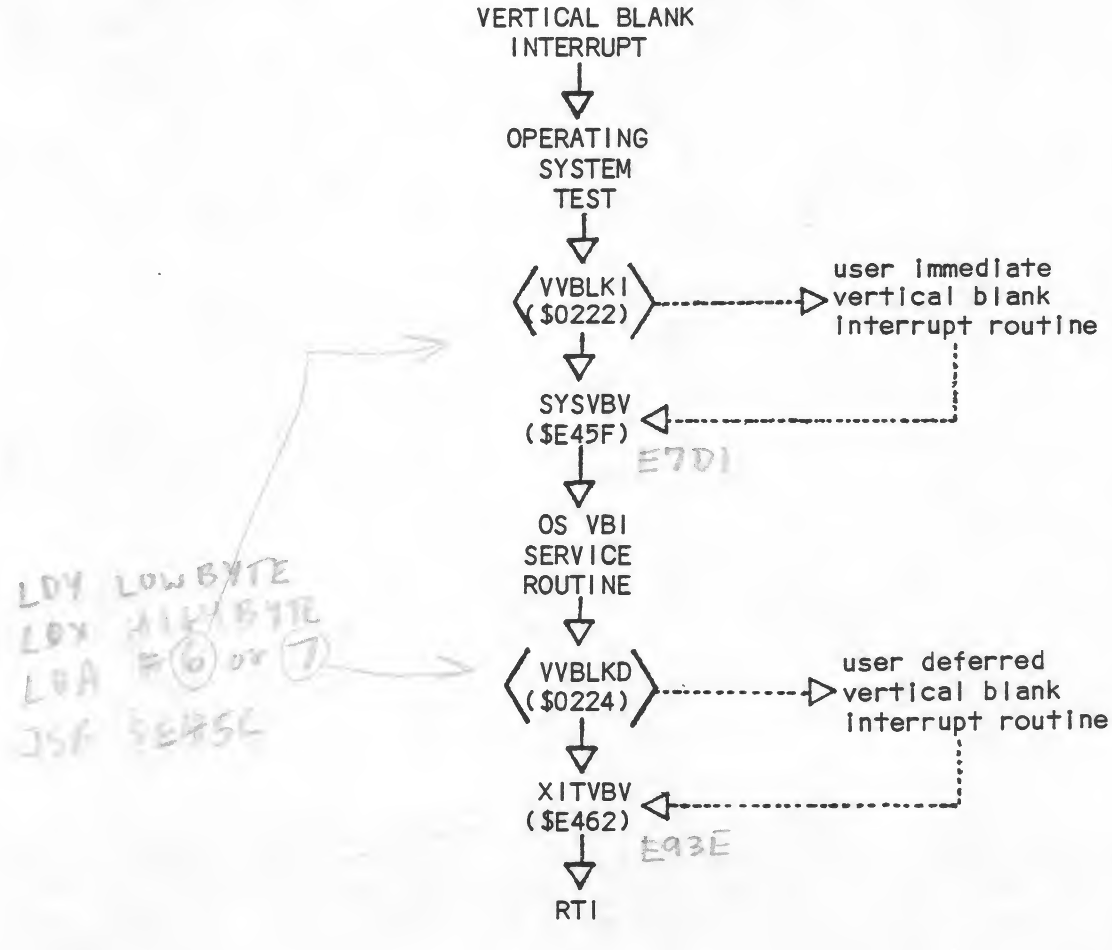
Impressionistic sounds affect our subconcious and our state of mind. This is due possibly to the fact that sounds, if present, are continuously entering our mind whether or not we are actively listening. On the other hand, if we are distracted from the TV set, we cease to concentrate on the picture and the image leaves our mind. Sound therefore offers the programmer a direct path to the user's mind - bypassing his thought processes and zeroing in on his emotions.

Even a very boring game can be made exciting by background mood sounds, jubilant success sounds, and chastizing but hopeful failure sounds. These can make a game much more popular, although they takes a great deal of effort to develop.

APPENDIX I
VERTICAL BLANK INTERRUPTS

The ATARI 400/800™ provides a number of interrupts which can be of great value. This appendix will cover vertical blank interrupts. These interrupts are non-maskable interrupts which occur every 60th of a second during the vertical blank time of the television display. They have a wide variety of uses.

At the beginning of vertical blank ANTIC pulls down the NMI line on the 6502. The 6502 then vectors to an NMI service routine which determines the source of the interrupt. If it is a vertical blank interrupt, the 6502 pushes its A, X, and Y registers onto the stack and jumps through the immediate vertical blank vector (VVBLKI) located at $0222. This vector normally points to the OS vertical blank interrupt service routine at $E45F. This routine terminates by jumping through the deferred vertical blank interrupt vector (VVBLKD) at $0224. Normally this vector points to a simple interrupt termination routine at $E462. Figure I.1 illustrates this process.



```
LDY LOWBYTE
LDX HIGHBYTE
LDA #6 or 7
JSR $E45C
```

Figure I.1
normal vertical blank interrupt execution
(and how to steal it)

These two vectors were put into RAM to allow the programmer to trap the interrupt service routine and use the 60 Hertz interrupt for her own purposes. The procedure to use them is rather simple. First decide whether the vertical blank interrupt (VBI) routine is to be an immediate VBI or a deferred VBI. In many cases it makes little difference which is chosen. There are only a few cases where it matters. The first case arises when your VBI routine reads or writes to registers which are shadowed by the OS VBI routine For example, it may be necessary to write to the hardware registers after the OS VBI routine has written to them so as to have the last word, so to speak.

The second case arises when your VBI routine consumes too much processor time. The OS VBI routine may be delayed beyond the end of the vertical blank period. This in turn may cause some graphics registers to be changed while the beam is tracing on the screen. The result may be unsightly. If this is the case, your VBI routine should be placed as a deferred VBI routine. Your time limit for immediate VBI routines is about 3800 machine cycles; for deferred VBI routines it is about 20,000 cycles. However, many of these 20,000 machine cycles are executed while the electron beam is being drawn, so graphics operations should not be executed in deferred VBI routines. Furthermore, display list interrupt execution time comes out of the time available for this processing. Remember also that VBI processing time comes out of the mainline execution time.

The third case arises when your own vertical blank interrupt must be mixed with time-critical I/O such as disk or cassette I/O. The OS vertical blank interrupt routine has two stages, a critical and a non-critical stage. During time critical I/O, the OS VBI routine aborts after stage one processing is complete. If you do not wish your own VBI routine to be disabled during time-critical I/O, you must define it as an immediate VBI routine. The delays you thereby create may interfere with time-critical I/O. That's your problem.

Once you have decided whether your VBI routine should be immediate or deferred, you must place the routine in memory (page six is an excellent place), link its termination to the regular VBI processing, and modify the appropriate OS RAM vector to point to it. Terminate an immediate VBI routine with a JMP to $E45F. Terminate a deferred VBI routine with a JMP to $E462. If you desire to bypass the OS VBI routine entirely (and so save some processing time), terminate the immediate VBI routine with a JMP to $E462.

A common problem with interrupts on 8-bit micros arises when you try to change the vector to the interrupt. Vectors are two-byte quantities; it takes two store instructions to change them. There is a small chance that an interrupt will occur after the first byte has been changed but before the second byte has been changed. This would crash the system. The solution to this problem is provided by an OS routine called SETVBV at location $E45C. Load the 6502 Y-register with the low byte of the address, the X-register with the high byte of the address, and the accumulator with a 6 for immediate VBI or a 7 for deferred VBI. Then JSR SETVBV and the interrupt will be safely enabled. It will begin executing within one 60th of a second.

A wide variety of operations can be performed with 60 Hertz interrupts. First, screen manipulations can be done during the vertical blank to insure that transitions do not occur on screen Second, high speed regular screen manipulations can be performed. This is important in showing many types of animation. For example, the bubbles in the nuclear reactor program SCRAM™ must move at a regular pace. They must not speed up or slow down as other computational activities burden the 6502. The only way to assure the regularity of their motion is to perform the animation during a vertical blank interrupt.

Another function of vertical blank interrupts is for sound envelope generation. The sound registers in the ATARI 400/800™ allow control of frequency, volume, and distortion, but not duration. Duration can be controlled with a vertical blank interrupt by having the calling routine set a duration parameter Then the vertical blank interrupt routine decrements any nonzero duration parameter and turns off the sound when the duration parameter reaches zero. This technique can be used to control the volume of the sound and so give attack and decay envelopes to sounds. Control of frequency and distortion is possible with extended versions of this technique. These techniques can produce very intricate sound effects. Because the time resolution is only 1/60th of a second, VBI's are not useful for direct control of speaker amplitude.

Vertical blank interrupts are also useful for handling user inputs. These inputs require little computation but constant attention. A vertical blank interrupt allows the program to check for user input every 60th of a second without otherwise burdening the program. It is an ideal solution to the problem of maintaining computational continuity without ignoring the user.

Finally, vertical blank interrupts allow a crude form of multitasking to take place. A foreground program can run under the vertical blank interrupt while a background program runs in the mainline. As with any interrupt, careful separation of the databases for the two programs must be maintained. However, the power obtained may well be worth the effort.

APPENDIX II
HUMAN ENGINEERING

The ATARI Personal Computer System is first and foremost a consumer computer. The hardware was designed to make this computer easy for consumers to use. There are many hardware features which protect the consumer from inadvertent errors. Software written for this computer should reflect an equal concern for the frailties of the consumer. The average consumer is not stupid; he is unfamiliar with the conventions and traditions of the computer world. Once he understands a program he will use it well most of the time. Occasionally he will be careless and make mistakes. It is the programmer's responsibility to protect the consumer from his own mistakes.

The current state of software human engineering in the personal computer industry is dismal. A great many programs are being sold which contain very poor human engineering. The worst offenders are written by amateur programmers, but even software written at some of the largest firms shows occasional lapses in human engineering.

Human engineering is an art, not a science. It demands great technical skill but it also requires insight and feel. As such it is a highly subjective field devoid of absolutes. This appendix is the work of one hand, and so betrays the subjectivities of its author. A proper regard for the wide variety of opinions on the subject would have inflated this appendix beyond all reasonable limits of length. Furhermore, a complete presentation of all points of view would only confuse the reader with its many assertions, qualifications, counterpoints, and contradictions. I therefore chose the simpler task of presenting only my own point of view, giving weak lip service to the most serious objections. The result is contradictory enough to satisfy even the most academic of readers.

THE COMPUTER AS SENTIENT BEING

An instructive way of viewing the problem of human engineering is to cast the programmer as sorcerer, conjuring up an intelligent being, a homunculus, within the innards of the computer. This creature lacks physical embodiment, but possesses intellectual traits, specifically, the ability to process and organize information. The user of the program enters into a relationship with this homunculus. The two sentient beings think differently; the human's thought patterns are associative, integrated, and diffuse, while the program's thought processes are direct, analytical, and specific. These differences are complementary and productive because the homunculus can do well what the human cannot. Unfortunately, these differences also create a communications barrier between the human and the homunculus. They have so much to say to each other because they are so different, but because they are different they cannot communicate well. The central problem in good programming must therefore be to provide for better communications between the user and the homunculus. Sad to say, many programmers expend the greater part of their efforts on expanding and improving the processing power of their programs. This only produces a more intelligent being with no eyes to see and no mouth to speak.

The current crop of personal computers have attained throughputs which make them capable of sustaining programs intelligent enough to meet many of the average consumer's needs. The primary limiting factor is no longer clock speed or resident memory; the primary limiting factor is the thin pipeline connecting our now-intelligent homunculus with his human user. Each can process information rapidly and efficiently; only the narrow pipeline between them slows down the interaction.

COMMUNICATION BETWEEN HUMAN AND MACHINE

How can we widen the pipeline between the two thinkers? We must focus on the language with which they communicate. Like any language, a man-machine language is restricted by the physical means of expression available to the speakers. Because the computer and the human are physically different, their modes of expression are physically different. This forces us to create a language which is not bidirectional (as human languages are). Instead, a man-machine language will have two channels, an input channel and an output channel. Just as we study human language by first studying the sounds that the human vocal tract can generate, we begin by examining the physical components of the man-machine interface.

OUTPUT (FROM COMPUTER TO HUMAN)

There are two primary ouput channels from the computer to the user. The first is the television screen; the second is the television speaker. Fortunately, these are flexible devices which permit a broad range of expression. The main body of this book describes the features available from the computer's point of view. For the purposes of this appendix, it is more useful to discuss these devices in terms of the human point of view. Of the two devices (screen and speaker) the display screen is easily the more expressive and powerful device. The human eye is a more finely developed information gathering device than the human ear. In electrical engineering terms, it has more bandwidth than the ear. The eye can process three major forms of visual information: shapes, color, and animation.

Shapes

Shapes are an ideal means for presenting information to the human. The human retina is especially adept at recognizing shapes. The most direct use of shapes is for direct depiction of objects. If you want the program to tell the user about something, draw a picture of it. A picture is direct, obvious, and immediate.

The second use of shapes is for symbols. Some concepts in the human lexicon defy direct depiction. Concepts like love, infinity, and direction cannot be shown with pictures. They must instead be conveyed with symbols, such as a heart, a horizontal figure 8, or an arrow. These are a few of the many symbols that we all recognize and use. Sometimes you can create an ad hoc symbol for limited use in your program. Most people can pick up such an ad hoc symbol quite readily. Symbols are a compact way to express an idea but they should not be used in place of pictures unless compactness is essential. A symbol is an indirect expression; a picture is a direct expression. The picture conveys the idea more forcefully.

The third and most common use of shapes is for text. A letter is a symbol; we put letters together to form words. The language we thereby produce is extremely rich in its expressive power. Truly is it said, "If you can't say it, you don't know it." This expressive power is gained at a price: extreme indirection. The word that expresses an idea has no sensory or emotional connection with the idea. The human is forced to carry out extensive mental gymnastics to decipher the word. Of course, we do it so often that we have become quite fluent at translating strings of letters into ideas. We do not notice the effort. The important point is that the indirection detracts from the immediacy and forcefulness of the communication.

There is a school of thought that maintains that text is superior to graphics for communications purposes. The gist of the arguement is that text encourages freer use of the reader's rich imagination. The arguement does not satisfy me, for if the reader must use his imagination, he is supplying information that is not inherent in the communication itself. An equal exercise of imagination with graphics would provide even greater results. A more compelling arguement for text is that its indirection allows it to pack a considerable amount of information into a small space. The space constraints on any real communication make text's greater compactness valuable. Nevertheless, this does not make text superior to graphics; it makes text more economical. Graphics requires more space, time, memory, or money, but it also communicates better than text. To some extent, the choice between graphics and text is a matter of taste, and the taste of the buying public is beyond question. Compare the popularity of television with that of radio, or movies with books. Graphics beats text easily.

Color

Color is another vehicle for conveying information. It is less powerful than shape, and so normally plays a secondary role to shape in visual presentations. Its most frequent use is to differentiate between otherwise indistinguishable shapes. It also plays an important role in providing cues to the user. Good color can salvage an otherwise ambiguous shape. For example, a tree represented as a character must fit inside an 8x8 pixel grid. The grid is too small to draw a recognizable tree; however, by coloring the tree green, the image becomes much easier to recognize. Color is also useful for attracting attention or signalling important material. Hot colors attract attention. Color also provides aesthetic enhancement. Colored images are more pleasing to look at than black and white images.

Animation

I use the term "animation" here to designate any visual change. Animation includes changing colors, changing shapes, moving foreground objects, or moving the background. Animation's primary value is for showing dynamic processes. Indeed, graphic animation is the only way to successfully present highly active events. The value of animation is most forcefully demonstrated by a game like STAR RAIDERS™. Can you imagine what the game would be like without animation? For that matter, can you imagine what it would be like in pure text? The value of animation extends far beyond games. Animation allows the designer to clearly show dynamic, changing events. Animation is one of the major advantages that computers have over paper as an information technology. Finally, animation is very powerful in sensory terms. The human eye is organized to respond strongly to changes in the visual field. Animation can attract the eye's attention and increase the user's involvement in the program.

Sound

Graphics images must be looked at to have effect. Sound can reach the user even when the user is not paying direct attention to the sound. Sound therefore has great value as an annunciator or warning cue. A wide variety of beeps, tone, and grunts can be used to signal feedback to the user. Correct actions can be answered with a pleasant bell tone. Incorrect actions can be answered with a raspberry. Warning conditions can be noted with a honk.

Sound has a second use: providing realistic sound effects. Quality sound effects can greatly add to the impact of a program because the sound provides a second channel of information flow that is effective even when the user is visually occupied.

Sound is ill-suited for conveying straight factual information; most people do not have the aural acuity to distinguish fine tone differences. Sound is much more effective for conveying emotional states or responses. Most people have a large array of associations of sounds with emotional states. A descending sequence of notes implies deteriorating circumstances. An explosion sound denotes destruction. A fanfare announces an important arrival. Certain note sequences from widely recognized popular songs are immediately associated with particular feelings. For example, in ENERGY CZAR™, I used a funeral dirge to tell the user that his energy mismanagement had ruined America's energy situation, and a fragment of "Happy Days Are Here Again" to indicate success.

INPUT DEVICES (FROM HUMAN TO COMPUTER)

There are three input devices most commonly used with the Atari Personal Computer Sytem. These are the keyboard, joystick, and paddles.

Keyboard

The keyboard is easily the most powerful input device available to the designer. It has over fifty direct keystrokes immediately available. Use of the CONTROL and SHIFT keys more than doubles the number of distinguishable entries the user can make. The CAPS/LOWR and ATARI keys extend the expressive range of the keyboard even further. Thus, with a single keystroke the user can designate one of 125 commands. A pair of keystrokes can address more than 15,000 selections. Obviously, this device is very expressive; it can easily handle the communications needs of any program. For this reason the keyboard is the input device of choice among programmers.

While the strengths of the keyboard are undeniable, its weaknesses are seldom recognized. Its first weakness is that not many people know how to use it well. Programmers use keyboards heavily in their daily work and consequently they are fast typists. The average consumer is not so comfortable with a keyboard. He can easily hit the wrong key. The very existence of all those keys and the knowledge that one must strike the correct key is itself intimidating to most people.

A second weakness of the keyboard is its indirection. It is very hard to attach direct meaning to a keyboard. A keyboard has no obvious emotional or sensory significance. The new user has great difficulty linking to it. All work with the keyboard is symbolic, using buttons which are marked with symbols which are assigned meaning by the circumstances. The indirection of it all can be most confusing to the beginner. Keyboards also suffer from their natural association with text displays; I have already discussed the weaknesses of text as a medium for information transfer.

Another property of the keyboard that the designer must keep in mind is its digital nature. The keyboard is digital both in selection and in time. This of course provides some protection against errors. Because keystroke reading over time is not continous but digital, the keyboard is not well-suited to real-time applications. Since humans are real-time creatures, this is a weakness. The designer must realize that use of the keyboard will nudge him away from real-time interaction with his target user.

Paddles

Paddles are the only truly analog input devices readily available for the system. As such they suffer from the standard problem all analog input devices share: the requirement that the user make precise settings to get a result. Their angular resolution is poor, and thermal effects produce some jitter in even an untouched paddle's output.

Their primary value is two-fold. First, they are well-suited for choosing values of a one-dimensional variable. People can immediately pick up the idea that the paddle sweeps through all values, and pressing the trigger makes the selection known. Second, the user can sweep from one end of the spectrum to the other with a twist of the dial. This makes the entire spectrum of values immediately accessible to the user.

An important factor in the use of paddles is the creation of a closed input/output loop. In most input processes, it is desirable to echo inputs to the screen so that the user can verify the input he has entered. This echoing process creates a closed input/output loop. Information travels from the user to the input device to the computer to the screen to the user. Because the paddle has no absolute positions, echoing is essential.

Any set of inputs that can be meaningfully placed along a linear sequence can be addressed with a paddle. For example, menus can be addressed with a paddle. The sequence is from the top of the menu to the bottom. It is quite possible (but entirely unreasonable) to substitute a paddle for a keyboard. The paddle sweeps through the letters of the alphabet, with the current letter being addressed shown on the screen. Pressing the paddle trigger selects the letter. While the scheme would not produce any typing speed records, it is useful for children and the idea could be applied to other problems.

Joysticks

Joysticks are the simplest input devices available for the computer. They are very sturdy and so can be used in harsh environments. They contain only five switches. For this reason their expressive power is frequently underestimated. However, joysticks are surprisingly useful input devices. When used with a cursor, a joystick can address any point on the screen, making a selection with the red button. With proper screen layout, the joystick can thus provide a wide variety of control functions. I have used a joystick to control a nuclear reactor (SCRAM™) and run a wargame (EASTERN FRONT 1941).

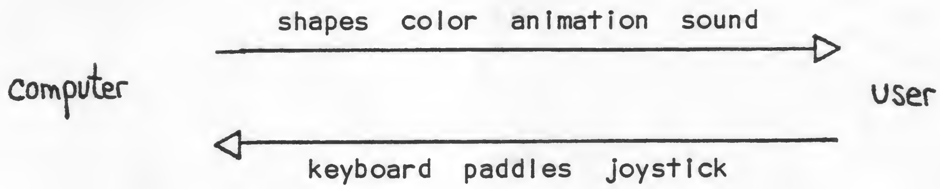
The key to the proper use of the joystick is the realization that the critical variable is not the selection of a switch, but the duration of time for which the switch is pressed. By controlling how long the switch is pressed, the user determines how far the cursor moves. This normally requires a constant velocity cursor. A constant velocity cursor introduces a difficult trade-off. If the cursor moves too fast, the user will have difficulty positioning it on the item of choice. If the cursor moves too slowly, the user will become impatient waiting for it to traverse long screen distances. One solution to this problem is the accelerating cursor. If the cursor starts moving slowly and accelerates, the user can have both fine positioning and high speed.

The real value of the joystick is its high tactility. The joystick involves the user in his inputs in a direct and sensory way. The tactility of the keyboard is not emotionally significant. A joystick makes sense---push up to go up, down to go down. If the cursor reflects this on the screen, the entire input process makes much more sense to the user.

Joysticks have their limitations. Although it is possible to press the joystick in a diagonal direction and get a correct reading of the direction, the directions are not distinct enough to allow diagonal entries as separate commands. Just as some words (e.g. "library", "February") are hard to enunciate clearly, so too are diagonal orders hard to enter distinctly. Thus, diagonal values should be avoided unless they are used in the pure geometrical sense: up on the joystick means up, right means right, and diagonally means diagonally.

SUMMARY OF COMMUNICATIONS ELEMENTS

I have discussed a number of features and devices which taken together constitute the elements of a language for interaction between the computer and the user. They are:



CONSTRUCTING A LANGUAGE

How do we assemble all of these elements into an effective language? To do so, we must first determine the major traits we expect of a good language. These are:

Completeness

The language must completely express all of the ideas that need to be communicated between the computer and the user. It need not express ideas internal to either thinker's thought processes. For example, the language used in STAR RAIDERS™ must express all concepts related to the control of the vessel and the combat situation. It need not express the player's anxiety or the flight path intentions of the Zylons. These concepts, while very germane to the entire game function, need not be communicated between user and computer.

Completeness is an obvious function of any language, one that all programmers recognize intuitively. Problems with completeness most often arise when the programmer must add functions to the program, functions which cannot be supported by the language she has created. This can be quite exasperating, for in many cases the additional functions are easily implemented in the program itself. The limiting factor is always the difficulty of adding new expressions to the I/O language.

Directness

Any new language is hard to learn. No user has time to waste in learning an unnecessarily florid language. The language a programmer creates for a program must be direct and to the point. It must rely as much as possible on communications conventions that the user already knows. It must be emotionally direct and obvious. For example, a CONTROL X keystroke is obscure. What does it mean? Perhaps it means that something should be destroyed; X implies elimination or negation. Perhaps it implies that something should be examined, expunged, exhumed, or something similar. If none of these possibilities are indeed the case, then the command is unacceptably indirect. Keyboards are notorious for creating this kind of problem.

Closure

Closure is the aspect of communications design that causes the greatest problems. The concept is best explained with an analogy. The user is at point A and wishes to use the program to get to point B. A poorly human-engineered program is like a tightrope stretched between points A and B. If the user knows exactly what to do and performs perfectly, she will succeed. More likely, she will slip and fall. Some programs try to help by providing a manual or internal warnings that tell the user what to do and what not to do. These are analogous to signs along the tightrope advising "BE CAREFUL" and "DON'T FALL". I have seen several programs that place signs underneath the tightrope, so that the user can at least see why she failed as she plummets. A somewhat better class of programs provide masks against illegal entries. These are equivalent to guardrails alongside the tightrope. These are much nicer, but they must be very well constructed to insure that the user does not thwart them. Some programs have nasty messages that bark at the errant user, warning her not to make certain entries. These are analogous to scowling monitors in the school halls, and are useful only for making an adult feel like a child. The ideal program is like a tunnel bored through solid rock. There is but one path, the path leading to success. The user has no options but to succeed.

The essence of closure is the narrowing of options, the elimination of possibilities, the placement of rock solid walls around the user. Good design is not an accumulative process of piling lots of features onto a basic architecture; good design requires the programmer to strip away minor features, petty options, and general trivia.

This thesis clashes with the values of many programmers. Programmers crave complete freedom to exercise power over the computer. Their most common complaint against a program is that it somehow restricts their options. Thus, deliberate advocacy of closure is met with shocked incredulity. Why would anyone be so foolish as to restrict the power of this wonderful tool?

The answer lies in the difference between the consumer and the programmer. The programmer devotes his life to the computer; the consumer is a casual aquaintance at best. The programmer uses the computer so heavily that it is cost-effective to take the time to learn to use a more powerful tool. The consumer does not have the time to lavish on the machine. He wants to get to point B as quickly as possible. He does not care for the fine points that occupy a programmer's life. Bells and whistles cherished by programmers are only obfuscatory trivia to him. You as a programmer may not share the consumer's values, but if you want to maintain your livelihood you damn well better cater to them.

Closure is obtained by creating inputs and outputs that do not admit illegal values. This is extremely difficult to do with a keyboard, for a keyboard always allows more entries than any real program would need. This is an excellent arguement against the use of the keyboard. A joystick is much better, because you can do so little with it. Because it can do so little, it is easier to conceptually exclude bad inputs. The ideal is achieved when all necessary options are expressible with the joystick, and no further options will fit. In this case the user cannot make a bad entry because it doesn't exist. More important, like Newspeak in Orwell's 1984, the user cannot even conceive bad thoughts because no words (inputs) for them even exist.

Closure is much more than masking out bad inputs. Masking makes bad inputs conceivable and expressible, but not functional. For examle, a keyboard might be used with the "M" key disabled because it is meaningless. The user can still see the key, he can imagine pressing it, and he can wonder what would happen if he did press it---all wasted effort. The user can waste even more time by pressing it and wondering why nothing happened. The waste is compounded by the programmer imagining the user doing all these wasteful things and putting in code to stop the symptoms without eliminating the disease. By contrast, a properly closed input structure uses an input device which can express only the entries necessary to running the program, and nothing more. The user can't waste time messing with something that isn't there.

The advantages that accrue when closure is properly applied are manifold. Code is tighter and runs faster because there need be no input error checking; such errors are obsolete in the new society, er, program. The user requires less time to learn the program and has fewer problems with it.

The primary problem with closure is the design effort that must be expended to achieve good closure. The entire relationship between the user and the program must be carefully analyzed to determine the minimum vocabulary necessary for the two to communicate. Numerous schemes of communication must be examined and discarded before the true minimum scheme is found. In the process, many bells and whistles that the programmer wanted to add will have to be eliminated. If the programmer objectively looks beyond her own values, she will often conclude that the bells and whistles are more clutter than chrome.

CONCLUSIONS

The design of the language of communication between the user and the program will be the most difficult part of the design process in consumer software. The designer must carefully weigh the capabilities of the machine and the needs of the user. He must precisely define the information that must flow between the two sentient beings. He must then design his language to maximize the clarity (not the quantity) of information flowing to the user while minimizing the effort the user must expend to communicate with the computer. His language must utilize the machine's features and devices effectively while maintaining its own completeness, directness, and closure.

SOME COMMON PROBLEMS IN HUMAN ENGINEERING

Having discussed the problems of human engineering in theoretical terms, I now turn to discuss specific application problems in human engineering. The list of problems is not exhaustive; it merely covers some of the problems common to almost all programs.

DELAY TIMES

Many programs require extensive computations. Indeed, almost all programs execute at some time computations that take more than a few seconds to perform. What does the user experience while these computations are executed? Too many programs simply stop the dialogue with the user for the duration of the computation. The user is left with an inactive screen and no sign of life from the computer. The computer does not respond to the user's inputs. If human engineering is created by the language of communication between the computer and the user, then this complete absence of communication can only be regarded as a total lack of human engineering. Leaving the user in the lurch like this is absolutely unforgiveable.

Separate processes

The best way to deal with the problem of reconciling computations with attentiveness is to separate the input process from the computational process. The user should be able to make inputs while the computations are proceeding. This is technically achievable; by using vertical blank interrupts (see Appendix I) the programmer can multitask input processing with mainline processing. The technique is used in EASTERN FRONT 1941. The real problem with the technique is that many problems are intrinsically sequential in nature. It is essential for the user to input a value or choice before the computation can proceed to the next step. This makes it difficult to separate input processing from the mainline processing. However, it is possible with clever design to perform anticipatory calculations that will determine intermediate values so that as soon as the critical data is entered, the result might be more quickly obtained. Application of such techniques can surely reduce the delay times that the user experiences.

Speed up the program

Another means of dealing with this problem is to speed up the program itself. Critical code can often be rewritten to decrease execution time. Proper nesting of loops (the loop with more iterations should be inside the loop with fewer iterations) can reduce execution time. Careful attention to the details of execution can yield further time reductions. Major gains can be made by converting BASIC to assembly language. Assembly is from 10 to 1000 times faster than BASIC. Assembly's advantage is greatest for memory move routines and graphics and least for floating point calculations. By masking out vertical blank interrupts, more 6502 execution time can be freed up for mainline processing. Other gains can be accomplished by reducing the DMA overhead ANTIC imposes. This can be done by going to a simple graphics mode (BASIC mode 3 is best). Shortening the display list is another way to reduce DMA costs. Turning off ANTIC altogether is a drastic route which only creates the additional problem of presenting the user with a blank screen.

Entertain the user

The third way to deal with delay times is to occupy the user during the computation. A countdown is one such method. The user seesDuntdown on the screen. When the countdown reaches zero, the program is back in business. Another way is to draw random graphics on the screen. The delay period should always start with a courteous message advising the user of the delay. It should also be terminated with a bell or other annunciator. You should not expect the user to keep his eyes on the screen for an arbitrary period of time. Entertaining the user during delays is a poor way to deal with delays that shouldn't have been there in the first place, but it's better than abandoning the user.

DEALING WITH BAD USER INPUTS

The most serious problem with present consumer software is the sloppy way that bad user inputs are handled. Good designs preclude this problem by providing input languages which do not make any bad entries available. As I pointed out earlier, this is most easily accomplished with a joystick. However, there are applications (primarily text-intensive ones) which require a keyboard. Furthermore, even joysticks occasionally introduce problems with user input. How are such bad inputs to be dealt with when they cannot be expunged? Several suggestions follow. It is imperative that any protection system be applied uniformly throughout the entire program. Once the user encounters protection, she will expect it in all cases. The lack of such protection creates a gap through which the user, thinking herself secure, will surely plunge.

Flag the error and suggest solution

The most desirable approach in this unpleasant situation is to flag the user's error on the screen in plain language and suggest a correct entry. Three things must be included the computer's response. First, the user's entry must be echoed back so he knows what he did that caused the problem. Second, the offending component of the entry must be clearly marked and explained so that the user knows why it is wrong. Third, an alternate legal entry must be suggested so that the user does not become frustrated by the feeling that he has encountered a brick wall. For example, an appropriate response to a bad keystroke entry might read thusly: "You pressed CONTROL-A, which is an autopsy request. I cannot peform autopsies on living people. I suggest you kill the subject first."

This method is obviously very expensive in terms of program size and programming time. That is the price one pays for bad design. There are less expensive and less effective methods.

Masking out bad keys

One common solution to keyboard input problems is to mask out all bad entries. If the user presses a bad key, nothing happens. No keyboard click is generated and no character appears on the screen. The program only hears what it wants to hear. This solution is secure in that it prevents program crashes, but it does not protect the user from confusion. The user would only press a button if he felt that it would do something for him. Masking out the button cannot correct the user's mistaken impression. It can only lead him to the conclusion that something is seriously wrong with his computer. We don't want to do this to our users.

A variant on this scheme is to add a nasty buzzer or raspberry to chastise the user for his foolishness. Indeed, some amateurish programs go so far as to heap textual abuse on the user. Such techniques are highly questionable. There may indeed be cases requiring dangerous keystroke entries which are guarded by fierce and nasty messages; such cases are quite rare. Corrective messages should always conform to high standards of civility.

Error messages

An even cheaper solution is to simply post an error message on the screen. The user is told only that he did something wrong. In many cases, the error message is cryptic and does not help the user in the least. ATARI BASIC is an extreme example of this. Error messages are provided by number only. This can be justified only when the program must operate under very tight memory constraints.

In most cases, the designer chooses to sacrifice human engineering features such as meaningful error messages for some additional technical power. As pointed out in the beginning of this appendix, we are reaching the stage in which additional technical power is no longer a limiting factor to consumers, but human engineering is a limiting factor. Thus, the trade-off is less justifiable.

Protection/power trade-offs

One objection to many human engineering features is that they slow down the user's ineraction with the computer. Programmers tire of incessant "ARE YOU SURE?" requests and similar restrictions. One solution to this problem is to provide variable protection/power ratios. For example, a program can default to a highly protected state on initialization. All entries are carefully checked and echoed to the user for confirmation. The user has an option to shed protection and work in high-speed mode. This option is not obvious from the screen---it is only described in the documentation. Thus, the intensive user can work at her pace and the casual user can have adequate protection.

MENUS AND SELECTION TECHNIQUES

Menus are standard devices for making the user aware of the options available. They are especially useful for beginning users. Command-oriented schemes preferred by programmers confuse beginners who cannot afford the time investment to learn the lexicon of commands used by a command-oriented program. There are several common problems associated with the use of menus. I shall discuss some of these.

Menu size

How many entries should be on a menu? The obvious upper limit is dictated by the size of the screen, but this limit is too large, for a BASIC mode 0 screen could hold up to 48 entries (24 lines with two choices per line). My guess is that 7 entries is the desired upper limit on menu size. This allows plenty of screen space to separate the entries, provide a menu title, and some sort of prompt.

Multiple menus

Frequently a program will require several menus to fully cover all of the options it offers. It is very important that multiple menus be organized in a clear manner. The user can easily get lost wandering around through such menu mazes. One way is to have a main menu that is prominently marked as such, and provide each secondary menu with an option to return to the main menu. Another way is to nest menus in a hierarchical structure. When using such methods, the programmer must provide color and sound cues to help the user ascertain his position in the menu structure. Each menu or menu level should have a distinctive note or color assigned to it. The note frequency should be associated with the position in the hierarchy.

Selection methods

Once the user has seen his options, how does he make his choice known to the computer? The most common way is to label each entry on the menu with a letter or number; the user makes his selection by pressing the corresponding key on the keyboard. This is a clumsy solution involving unnecessary indirection. There are a number of better methods. Most of them use the same basic scheme: a movable pointer addresses an option, and a trigger selects it. One scheme I have seen highlights the option being addressed in inverse video. The SELECT button changes the pointer to address the next menu selection, with full wraparound from the end of the menu to the beginning. The START button engages a menu option. Another program I have seen automatically rotated the pointer through the menu options; the user need only push a button at the correct moment when his desired option was being addressed. (I wasn't overly impressed by that method.) Paddles and joysticks are very well suited for menu selection. Either one can be used to sweep the pointer through the menu selections, with the red trigger button making the selection. My pet scheme for menu selection uses a cursor on a large scrolling menu. The user moves the cursor with a joystick. Signposts can direct her to different regions of the menu. She makes her selection by placing the cursor directly on top of an option and pressing the trigger button.

MANUALS VERSUS ON-BOARD TEXT

A common problem with menus, error messages, prompts, and other messages is that such material can easily consume a large amount of memory, memory that could well be used for other features. Such material could be placed in a reference document, but doing so would detract from the quality of the program's human engineering. The designer must decide how much material should go into the program and how much should be relegated to the manual. With disk-based programs it is possible to store some of the material on the diskette; this lessens the harshness of the trade-off. When the problem is approached only from the human engineering point of view, it is easily answered: all material should be included in the program, or at least on a diskette. Economic and technical considerations argue against this. It is my personal view that each technology should be used for the things it does best. While the computer can handle static text, its forte is dynamic information processing. Paper and ink handle static information more cheaply and often more clearly than a computer. I therefore prefer to put static information into a manual and let the program refer the user to the manual. I still include critical information within the program; my dividing line bends with local needs.

MEASURES OF SUCCESS

How can a designer determine the success of his human engineering? There are several indicators that provide valuable feedback. The first is the minimum length length of the manual. If you exclude background material and isolate only the material in the manual that is absolutely necessary to describe how to use the program, then the length of this material is a good measure of your human engineering. The more material, the worse you've done. A well-designed program should require very little explanation. This should not be construed as an arguement against proper documentation. Documentation should always describe the program in more detail than is absolutely necessary. A long, lavish manual is good; a program that demands such a manual is not.

Another measure is the amount of time that a first-time user expends to learn to use the program satisfactorily. Good programs can be used in a matter of minutes.

A third measure is the amount of thinking a user must do to use the program. A well-designed program should require no cognitive effort to use. This does not mean that the user does not think at all while using such a program. Rather, he thinks about the content of the program rather than the mechanics of the program. He should concentrate on what he is doing, not how he does it.

The well-engineered program eliminates mental distance between the user and the computer. The two thinking beings achieve a mental syntony, an intellectual communion.

APPENDIX III

ATARI CASSETTE OVERVIEW

This is a discussion of the ATARI 410™ Program Recorder. The following topics will be included:

1. How the cassette works - information on the hardware and software used to operate the cassette.

2. Cassette applications - how to mix audio and digital information to produce a very user oriented program.

HOW THE CASSETTE WORKS

1.1 RECORD STRUCTURE

Byte Definition:

The OS writes files in fixed-length blocks at 600 baud (physical bits/second). Asynchronous serial transmission is used to read and write data between the ATARI PCS and the ATARI 410. POKEY recognizes each data byte in this order: 1 start bit (space), eight data bits (0=space, 1=mark), then one stop bit (mark). A byte is sent/received least significant bit first.

The frequency used to represent a mark is 5327 Hz. For a space the frequency is 3995 Hz. The data byte format is as follows:

```
______      _0_     _2_     _4_     _6_     __B_______ <______ MARK
      |    |   |   |   |   |   |   |   |   |
      |_A_|    |_1_|   |_3_|   |_5_|   |_7_|           <______ SPACE
```

```
A    = Start Bit (Space)
0-7  = Data Bits
B    = Stop Bit (Mark)
```

Record Definition:

Records are 132 bytes long. A record is broken down in the following way: 2 marker characters for speed measurement, a control byte, 128 data bytes, and the checksum byte. The record format is shown below:

```
 _______________
|               |
|0 1 0 1 0 1 0 1|          1st MARKER
|_______________|                           (For Speed
|               |                            Measurement)
|0 1 0 1 0 1 0 1|          2nd MARKER
|_______________|
|               |
| CONTROL BYTE  |
|_______________|
|      128      |
=     DATA      =
|     BYTES     |
|_______________|
|               |
|   CHECKSUM    |
|_______________|
```

1st and 2nd MARKER:

Each marker character is a 55 (HEX). Including start and stop bits, each marker is 10 bits long. Ideally, there should be no blank tape between the markers and the subsequent data.

Speed Measurement:

The purpose of the marker characters is to adjust the baud rate.

The input baud rate is assumed to be a nominal 600 baud. This is adjusted, however, by the SIO routine to account for drive motor variations, stretched tape, etc. Once the true receive baud rate is calculated, the hardware is adjusted accordingly. Input baud rates ranging from 318 to 1407 baud can theoretically be handled using this technique.

The OS checks the tape speed in the following manner: The software looks at the POKEY Serial-In bit continuously. Looking for a start (0 bit) which signifies the beginning of a record. When it finds one, the OS stores the current frame counter by saving the ANTIC VCOUNT (vertical screen counter). Continuing to look directly at the Serial-In bit, the OS counts the twenty bits (end of the 2 markers), then uses VCOUNT and the frame counter to determine the elapsed time. The baud rate to use is derived from the result. This is done for each record.

Control Byte:

The control byte contains one of three values:

$FC indicates the record is a full data record (128 bytes).

$FA indicates the record is a partially full data record; fewer than 128 bytes were supplied by the user. This case may occur only in the record prior to the end-of-file. The actual number of data bytes, 1 to 127, is stored in the last data byte prior to the checksum; i.e. the 128th data byte.

$FE indicates the record is an end-of-file record and is followed by 128 zero bytes.

Checksum:

The checksum is generated and checked by the SIO routine, but is not contained in the cassette handler's I/O buffer CASBUF [03FD].

The checksum is a single byte sum of all the other bytes in the record, including the two markers. The checksum is computed with end-around carry. As each byte is added into the sum, the carry bit is also added in.

```
┌──►  Partial Sum
│  +  Data Byte
│  +  Carry
│  ─────────────
└──   Result
```

1.2 TIMING

1.2.1 INTER-RECORD GAP (IRG)

As was mentioned in section 1.1 each record consists of 132 data bytes including the checksum byte. In order to distinguish one record from another, the cassette handler adds a Pre-Record Write Tone (PRWT) and Post-Record Gap (PRG). PRWT and PRG are both pure mark tone. The Inter-Record Gap (IRG) between any two records thus consists of the PRG of the first record followed by the PRWT of the second record. The layout of the records and gaps is as follows:

PRWT	MARKER	DATA	PRG	PRWT	MARKER	DATA	PRG

← RECORD 1 →   ← RECORD 2 →

1.2.2 NORMAL IRG MODE & SHORT IRG MODE

The length of PRWT and PRG are dependent upon the Write Open mode. There are 2 types of IRG modes: Normal IRG mode and Short IRG mode.

When a file is opened the most significant bit of AUX2 specifies the mode. On subsequent output or input, the cassette handler executes the READ/WRITE in either mode based on the MSB of the AUX2 byte:

	7							0
AUX2	C							

C = 1 indicates that the cassette is to be read/written in Short IRG mode. (Continuous mode)
C = 0 indicates Normal IRG mode.

Normal IRG Mode:

This mode is used for a READ interleaved with processing; i.e. the tape always comes to a stop after each record is read. If the computer "STOPS" the tape and gets its processing done fast enough, the next READ may occur so quickly that the cassette deck may see only a slight dip in the control line.

Short IRG Mode:

In this mode the tape is not stopped between records, either when being written or during readback.

On readback, the program must issue a READ for each record before it passes the read head. The only common use of this mode so far is storage of BASIC programs in internal (tokenized) form where, on readback, BASIC has nothing more to do with the data than put it in RAM. The special BASIC commands "CSAVE" and "CLOAD" specify this mode.

There can be a potential problem with this. The software that writes the tape must allow long enough gaps, so the beginning of records are not missed on readback.

1.2.3 TIMING STRUCTURE

The timings for each of the inter-record gaps are as follows:

```
NORMAL IRG PRWT = 3 seconds of mark tone.
SHORT IRG PRWT  = 0.25 seconds of mark tone.

NORMAL IRG PRG  = Up to 1 second of unknown tones.
SHORT IRG PRG   = From 0 to N seconds of unknown tones, where N
                  is dependent upon user program timing.
```

Each record is written with the following timing: once the motor starts and the Pre-Record Write Tone (PRWT) is written, the duration of the tone depends on the above format. The record follows, then the Post-Write Gap (PRG) is written. The motor is then stopped for Normal mode, but continues writing mark for Short IRG mode.

Note that for the Normal IRG mode, the tape will contain a section of unknown data because of stopping and restarting the motor. (Up to 1 second of travel is possible, depending on the cassette machine.) This unknown data may be garbage data left previously on the tape.

1.2.4 NOISY I/O FEATURE

The Noisy I/O feature is useful for determining the success of reading the tape, particularly with CLOAD. Marks and spaces use different sound frequencies and one quickly learns the good and bad sounds the OS makes.

1.3 FILE STRUCTURE

A file consists of the following three elements:

1) A 20 second leader of the mark tone.
2) Any number of data records.
3) End-Of-File.

When the file is opened (output), the OS starts by writing a mark leader of 20 seconds, the OS then returns to the caller, but leaves the tape running and writing marks.

The WRITE/READ timeout counter is set for about 35 seconds as the OS returns. If the timeout occurs before the first record is written, the tape will stop, leaving a gap between the open leader and the first record leader.

1.4 TAPE STRUCTURE

There are 2 sides to each tape. Each side has 2 tracks, one for audio and the other one for digital recording. This way the tape can be recorded in both directions. Following is a flat view of the tape:

```
--------------------------------------------------------------------------------
           ............................................
               ////////AUDIO TRACK////////                    LEFT TRACK
           ............................................
SIDE
 A         ............................................
               ////////DIGITAL TRACK////////                  RIGHT TRACK
           ............................................

--------------------------------------------------------------------------------
           ............................................
               ////////DIGITAL TRACK////////                  RIGHT TRACK
           ............................................
SIDE
 B         ............................................
               ////////AUDIO TRACK////////                    LEFT TRACK
           ............................................

--------------------------------------------------------------------------------
```

Tapes are recorded in 1/4 track stereo format at 1 7/8 inches per second (IPS). Note that the Atari 800 utilizes a tape deck that has a stereo head configuration (not a single or mono type).

1.5 CASSETTE BOOT

The Cassette Boot program can be booted from the cassette at power up time as part of the system initialization.

System initialization performs functions such as zeroing all of the hardware registers, clearing RAM, setting flags and so on.

After all the resident handlers are brought in, if the 'START' key is pressed, the Cassette Boot request flag CKEY [004A] is set. If the Cassette Boot request flag is set, then a Cassette Boot operation is attempted.

The following requirements must be met in order to boot from the cassette:

1) The operator must press the 'START' key as power is applied to the system.

2) A cassette tape with a proper boot format file must be installed in the cassette drive, and the 'PLAY' button on the recorder must be pressed.

3) The cassette file must have been created in Short IRG mode.

4) When the audio prompt occurs, the operator must press a key on the keyboard.

If all of these conditions are met, the OS will READ the boot file from the cassette and then transfer control to the software that was read in. The Cassette Boot process is given in more detail below.

1) READ the first cassette record to the cassette buffer.

2) Extract information from the first 6 bytes. The first 6 bytes of a Cassette Boot file are formatted as shown below:

IGNORED		1st BYTE
# OF RECORDS		
MEMORY ADDRESS	LO	
TO START LOAD	HI	
INIT	LO	
ADDRESS	HI	6th BYTE

1ST BYTE: is not used by the Cassette Boot process.

2ND BYTE: contains the number of 128 byte cassette records to be read as part of the boot process (including the record containing this information). This number may range from 1 to 255, with 0 meaning 256.

3RD and 4TH BYTES: contain the address (LO,HI) at which to start loading the first byte of the file.

5TH and 6TH BYTES: contain the address (LO,HI) to which control is transferred after the boot process is complete. Pressing the [S/RESET] key will also transfer control to this address assuming that the boot process is complete.

When step 2 is complete, the Cassette Boot program will have:

A) saved # of records to boot.
B) saved the load address.
C) saved the initialization address in CASINI [02,03].

3) Move the record just read to the load address specified.

4) READ the remaining records directly to the load area.

5) JSR to the load address +6 where a multi-stage boot process may continue. The carry bit will indicate the success of the operation (carry set = error, carry reset = success) on return.

6) JSR indirectly through CASINI for initialization of the application. The application should put its starting address into DOSVEC [0A,0B] during initialization, and then return.

7) JMP indirectly through DOSVEC to transfer control to the application.

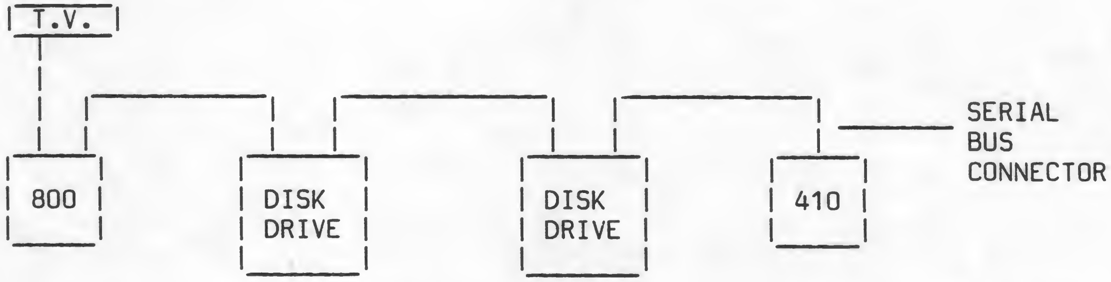
Pressing the [S/RESET] key after the application is fully booted will cause steps 6 and 7 to be repeated.

CASSETTE APPLICATIONS

This section covers how to utilize the Atari cassette system.

2.1 HOW TO CONFIGURE THE CASSETTE SYSTEM

Most serial bus devices have two identical connectors: one is a serial bus input and the other a serial bus extender. Using these connectors peripherals may be "Daisy Chained" simply be cabling them together in a sequential fashion like the following diagram:



However, the cassette does not conform to the protocol of the other peripherals that use the serial bus. The cassette must be the last device on the serial bus because it does not have a serial bus extender connector as the other peripherals do. The lack of a bus extender assures that there is never more than one cassette drive connected to the system. The system cannot sense the absence or presence of the cassette drive, so it may be connected and disconnected at will.

Whenever there is a need to open a cassette file for reading or writing, the user will have to follow the following instructions:

INPUT (DATA FROM 410 TO 800): When the cassette is opened for input, a single audible tone is generated using the keyboard speaker. If the cassette is ready (power on, serial bus cable connected, tape cued to start of file), the user must depress the 'PLAY' button on the cassette and any 800 keyboard key (except [BREAK]) to initiate tape reading.

OUTPUT (DATA FROM 800 TO 410): When the cassette is opened for output, two separate audible tones are generated using the keyboard speaker. If the cassette is ready (as previously described), the user must simultaneously press the 'PLAY' and 'RECORD' buttons on the cassette, and then press any keyboard key (except [BREAK]) to initiate writing the tape.

2.2 SAVING AND LOADING DIGITAL PROGRAMS

Concept:

The following technique saves the digital data directly from the computer through its I/O port of either the 410 or the Atari Lab Machine which uses 1/4 inch tape recorded at 7 1/2 inches per second.

FOR BASIC:

```
FORMAT:   CSAVE
          100 CSAVE
```

This command is usually used in direct mode to save a RAM-resident program onto cassette tape. 'CSAVE' writes the tokenized version of the program to the 410.

```
FORMAT:   CLOAD
          100 CLOAD
```

This command can be used in either direct or deferred mode to read programs from cassette tape into RAM for execution.

FOR ASSEMBLY LANGUAGE:

```
SOURCE PROGRAM
FORMAT:   LIST#C:[,XX,YY]
```

This command is used to write assembly source code. The items in the optional brackets [,XX,YY] mean to transfer only lines XX to YY to cassette. If line numbers are not provided the whole program is listed to cassette.

```
FORMAT:   ENTER#C:
```

This command reads source code from the cassette.

```
OBJECT PROGRAM
FORMAT:   SAVE#C:<XXXX,YYYY
```

The contents of a block of memory, locations XXXX to YYYY, is saved onto cassette.

```
FORMAT:   LOAD#C:
```

This command will load memory with the material that was previously saved. The range of memory locations that are filled will be the same as those given in the original save command.

2.3 SAVING DIGITAL PROGRAMS WITH AUDIO AS BACKGROUND

Concept:

This recording technique does not allow any program control over the audio. The audio plays purely as background to help time pass during the monotonous loading process.

STEP 1: Follow the digital writing instructions indicated in 2.2 for BASIC and Assembly programs; except, this time ATARI standard cassette tape (1 7/8 inches per second) is not used. Because it is hard later for an individual to record audio onto 410, we have to use the ATARI recording lab machine, which uses 7 1/2 inches per second master tape. The lab machine is a much more sophisticated recording machine able to record data onto a specified track.

On the lab machine, the recording mode is switched to "ON" for the right track, so digital is saved onto the right track of the 7 1/2 inch tape.

STEP 2: Do STEP 1 for audio recording, except first rewind the tape to the beginning of the program then switch the recording mode to "ON" for left track. This way the audio is recorded onto the left track of the 7 1/2 inch tape.

2.4 DIGITAL PROGRAMS, AUDIO, SYNC MARK, AND SCREEN MANAGEMENT

Sync Mark Concept:

There is no efficient way for the program to detect an audio segment when the cassette is playing. In order to solve the synchronization problem, Sync Mark is used to carry the signal to inform the program that an audio segment has been played (an audio segment can be either a piece of music or an instruction, depending on the application).

More precisely, since audio data has no record structure, Sync Mark recorded on the digital track is more or less like End-Of-Record Mark for audio. For example, once the program senses the Sync Mark, the program can decide what to do next like stop the cassette motor for lengthy processing, or continue to play the next audio segment.

STEP 1: The programmer figures out an audio script for "FROG". The script is like this:

(MUSIC) TODAY I AM GOING TO TELL YOU A FAIRY-TALE NAMED "THE PRINCESS AND THE FROG". IT IS A SWEET STORY SO DON'T GO AWAY. /

(MUSIC) BEFORE I START MY STORY, I WOULD LIKE TO KNOW WHO I AM TALKING TO. WHAT IS YOUR NAME? TYPE YOUR NAME AND HIT CARRIAGE RETURN. (PAUSE)

(MUSIC) NOW, LET'S START THE STORY. ONCE UPON A TIME, THERE WAS THIS BEAUTIFUL PRINCESS LIVING IN A CASTLE AND HER NAME WAS YYYY. /

(MUSIC) ON A CLEAR AND BEAUTIFUL DAY, THE PRINCESS WAS WALKING ALONG THE ..../

REMARK:
- "/" means the program is checking for a sync mark. It is best if the speaker pause about 1/2 second here before continuing to the next segment of the audio script.
- "PAUSE" is to indicate that the speaker pauses about 1 second here to allow time for the stopping and starting of the cassette motor. Each audio segment should be at least 10 to 30 seconds long, because too many closely spaced Sync Marks can confuse the computer.

STEP 2: It is suggested that before coding begins, the programmer draft a general plan for the program indicating the relationship between screen (CPU) and audio.

EXAMPLE: The following example (see page III-13) illustrates how a programmer should create a cassette containing a program which has control over an audio track. The example is called "FROG":

STEP 3: The programmer can start coding the program called "FROG", and it will look something like this:

```
 10 REM PROGRAM "FROG" TO DEMONSTRATE SYNCHRONIZATION
 20 REM OF AUDIO WITH DIGITAL FOR THE CASSETTE SYSTEM
 30 REM
 40 DIM IN$(20)
 50 POKE 54018,52:REM TURN ON MOTOR
 60 GRAPHICS 1
 70 PRINT #6;"THE PRINCESS AND THE FROG":PRINT #6;....:REM
      SET UP THE SCREEN FOR EVENT 2.
 80 GOSUB 1000:REM CHECK SYNC MARK, MAKE SURE THE INTRODUCTION
      IS SAID.
100 POSITION X,Y:PRINT #6;"YOUR NAME?":REM FOR EVENT 4
105 GOSUB 1000:REM EVENT 5
110 POKE 54018,60:REM STOP MOTOR FOR USER INPUT
120 INPUT IN$:REM WAIT FOR THE USER'S NAME
130 POKE 54018,52
135 PRINT #6,CHR$(125):REM CLEAR THE SCREEN
140 POSITION X,Y:PRINT #6;IN$:PRINT #6;....:REM DISPLAY
      SCREEN FOR EVENT 10
150 GODUB 1000:REM MAKE SURE SPEECH FOR EVENT 10 IS FINISHED
160 PRINT #6;....:REM READY FOR EVENT 12
```

"FROG"

EVENT	AUDIO	SCREEN	CHECK	MOTOR
			SYNC MARK	MODE
1				ON
2	'TODAY I AM' 'GOING TO..'	THE PRINCESS & THE FROG GRAPHIC		
3			YES	
4	BEFORE I....'	THE PRINCESS & THE FROG GRAPHIC YOUR NAME?XXXX		
5			YES	
6				STOP
7		WAIT TILL AN INPUT IS RECOGNIZED		
8				START
9		CLEAR THE SCREEN		
10	'NOW LET'S....'	XXXX GRAPHIC		
11			YES	
12	'ON A CLEAR..'	GRAPHIC		
13				

ROUTINE TO CHECK SYNC MARK: On the tape, non-sync is represented by "MARK" and Sync Mark is represented by "SPACE". (Space is a "0" frequency, it is a lower pitch sound than a Mark which is a "1" frequency. As mentioned before, Mark frequency is 5327 Hz, Space is 3995 Hz). The Check Sync Mark routine continuously watches for a "SPACE" from the serial port. The routine looks like this:

```
1000 IF INT(PEEK(53775)/32+0.5)=INT(PEEK(53775)/32)
     THEN RETURN: REM CHECK THE 5TH BIT OF EACH
                  INCOMING BYTE.  IF IT IS "0"THEN
                  THE SYNC SPACE IS FOUND.
1010 GOTO 1000
```

ROUTINE TO CONTROL THE MOTOR: The program can turn the cassette motor on and off by poking location 54018 with the data given below:

```
 ON:   POKE 54018,52
OFF:   POKE 54018,60
```

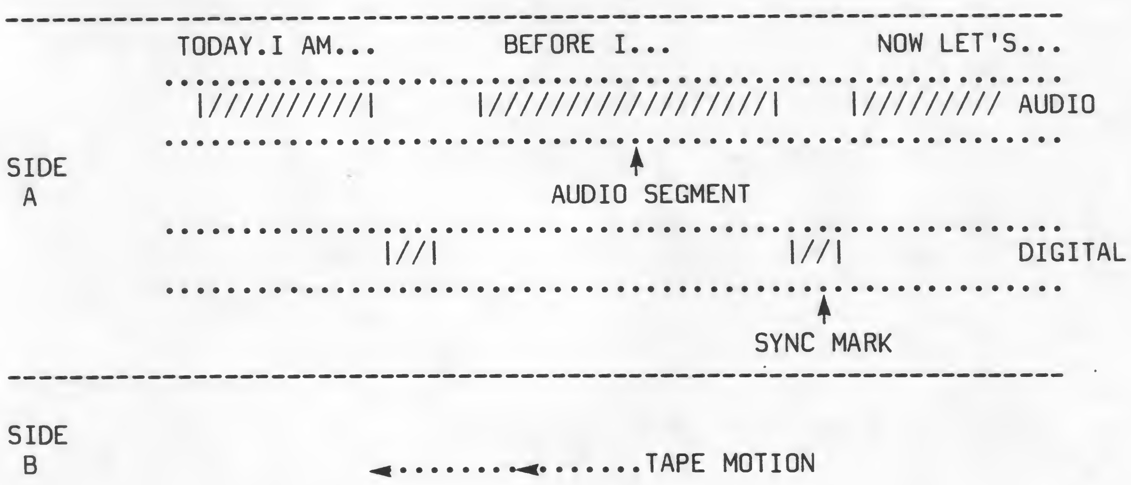
STEP 4: After the audio script has been roughly written, the programmer should estimate the time and the tape length required for the designed audio script (including pauses) and program. If the tape length required is too long for one cassette, then either the script or the program will have to be modified to fit into one cassette.

STEP 5: Save the program to a master tape, for example "MASTER 1".

STEP 6: With the audio script the voice is taped with pauses on another master tape, "MASTER 2".

STEP 7: After "MASTER 1" and "MASTER 2" are produced, these 2 master tapes are merged to produce another master tape called "MASTER 3". "MASTER 3" has the program recorded first, and the audio spliced on the end. Three recording lab machines are needed for this procedure. Make 2 copies of "MASTER 3".

STEP 8: Load the Sync Mark program into the Atari 800. The purpose of this program is to write continuous Sync Mark ("0" frequency) onto the digital track. The Sync Mark informs the program that an audio segment has been played. Whenever there is a pause indicated on the audio script, a Sync Mark is needed at that place. The finished tape with audio and sync would be as follows:



The Sync Mark program looks like this:

```
 10 REM PUSH "START" CONSOL KEY TO
 20 REM ADD THE SYNC MARK ONTO THE TAPE
 30 REM
 40 REM
 50 IO=53760 : CONSOLE=53279 : CASS=54018
100 FOR I=0 TO 8
110 READ J : POKE IO+I,J
120 NEXT I
125 REM THE FOR LOOP SETS THE AUDIO FREQUENCY & CHANNEL
130 DATA 5,160,7,160,5,160,7,160,0
140 REM
150 REM I/O IS SETUP; NOW START THE CASSETTE
160 POKE CASS, 52
200 POKE CONSOLE,8
210 IF PEEK(CONSOLE) <> 7 THEN 230:REM CONSOLE=7 MEANS WRITE
     MARK,
220 POKE IO+15,11: GOTO 200: REM CONSOLER KEYS NOT PRESSED
230 POKE IO+15,128+11: GOTO 200: REM IF CONSOLE <> 7 WRITE
     "SPACE"
```

STEP 9: Mount both "MASTER 3" tapes in two independent recording machines and rewind both tapes to the splice of program and audio. Configure one recording machine to one Atari 800 with Sync Mark program loaded. This recording machine is prepared for recording Sync Mark on the digital track. The other recording machine will play back the audio recorded earlier.

STEP 10: Type "RUN" to start the Sync Mark program. At the same time start the recording machines, one for recording, another one for playback. Listen to the audio and hit the "START" key whenever it is indicated by a pause in the audio script.

STEP 11: Now the tape is done, with the program recorded followed by the audio and Sync Mark recording. The finished tape is ready for mass production.

2.5 DISABLING THE BREAK KEY

It is suggested that the programmer disable the BREAK key. This prevents the cassette program from failing then the user accidently hits "BREAK". The OS will not recover a partial record, unless the user can rewind to the lost record. The disable BREAK key routine looks like this:

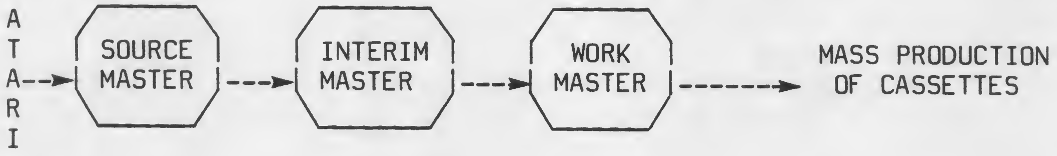
```
4000 X= PEEK(16): IF X 128 THEN 4020
4010 POKE 16,X-128: POKE 53774,X-128
4020 RETURN
```

The disable routine should be called whenever there is a change of graphics mode or any screen open call.

2.6 MASS PRODUCTION

The programmer produces one or more "MASTER TAPES" according to the recording techniques discussed in sections 2.2, 2.3, and 2.4. All Atari Masters are recorded on open reel 1/4 track, 1/4 inch tape recorded at 7 1/2 inches per second. The "MASTER TAPE" is supplied to the duplicator as a "SOURCE MASTER".

The duplicator will take the "SOURCE MASTER" to make a "WORK MASTER" for the final cassette mass production. The released product will be third generation from the original. The following is a flow of the process:

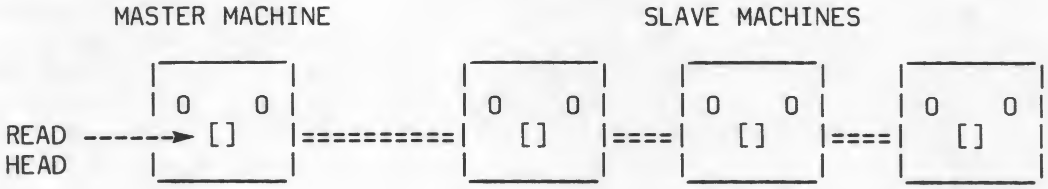


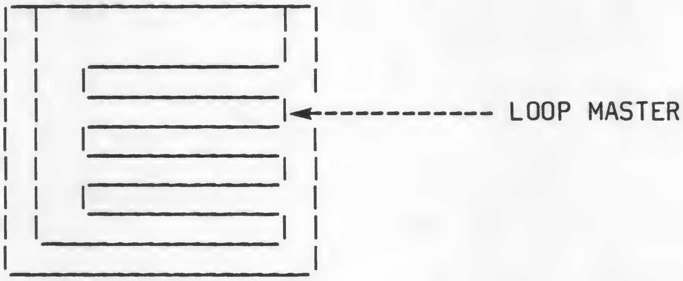
"INTERIM MASTER" is recommended for the duplicator, because the "WORK MASTER" may be destroyed or worn from excessive use. The "SOURCE MASTER" should be reserved only for emergency need. The "INTERIM MASTER" is the backup copy for the "WORK MASTER".

2.6.1 <u>MASS PRODUCTION OF CASSETTES</u>

At present, ATARI prefers the "BIN LOOP" method for mass production: The "WORK MASTER" is copied to produce a "LOOP MASTER". The LOOP MASTER may be on 1/4", 1/2", or any tape width. The BIN LOOP is spliced into a CONTINUOUS LOOP with a short clear leader at the splice. It is placed in a high-speed loop master machine which has one or more "SLAVE" machines. The configuration is like this:

ATARI CASSETTE





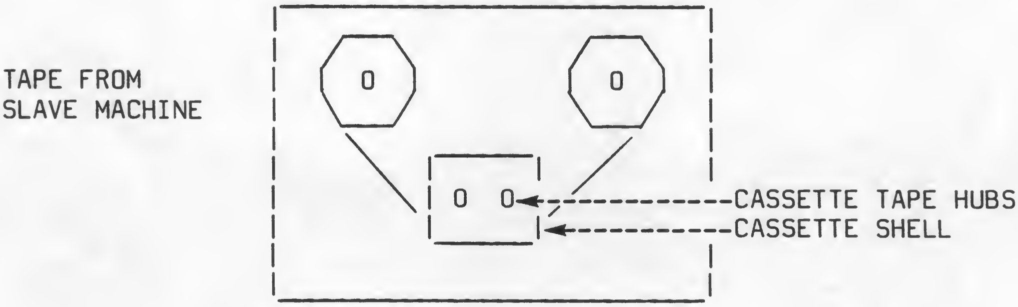
The "LOOP MASTER" is repeatedly read. If the duplicator wants to produce 100 cassettes, for example, the length of the tape on the "SLAVE MACHINE" is measured to the length of the program multiplied by 100. There is a counter on the "MASTER" machine and it is set to 100.

As the "LOOP MASTER" is continuously read, the data (all four tracks) is copied onto the "SLAVE MACHINE" tape.

As the clear section in the LOOP MASTER is sensed, the "MASTER" machine produces a "CUTTING TONE" which is recorded on one or more tracks on the SLAVE MACHINE tapes. The counter will then increase by one.

Each finished tape from the "SLAVE MACHINE" has 100 recorded programs with 100 CUTTING TONES recorded. It is fed into an automatic loading machine which winds the tape into C-Zero cassette shells. The configuration is like this:

LOADER



The cassette shells come with a small loop of leader which is bound to the cassette tape hubs. The loader pulls the leader from the shell, cuts it, and splices the end of the slave machine tape to the leader. The tape hub is used to wind the tape into the shell until the cutting tone is sensed. The slave machine tape is then cut and spliced to the leader on the other hub.

The cassette shell is removed either manually or mechanically from the loader and the tape in the cassette shell is fully wound. The next cassette shell is loaded by the same process.

2.6.2 QC TESTING

Any time that a production run is created, samples must be taken from it and verified before it is approved and released.

The QC testing is done normally by taking the first and the last cassette produced. Atari must receive at least 10 samples from each mass production for each master released.

APPENDIX IV

TELEVISION ARTIFACTS

This section discusses how to get multiple colors out of a single color graphics mode through the use of television artifacts.

The ANTIC modes with which this can be accomplished are 2,3, and 15 ANTIC mode 2 corresponds to BASIC mode 0, ANTIC mode 15 is BASIC mode 8, and ANTIC mode 3 has no corresponding BASIC mode. Each of these modes has a pixel resolution of one half color clock by one scan line. They are generally considered to have one color and two luminances. With the use of artifacts, pixels of four different colors can be displayed on the screen in each of these modes.

The term TV artifacts refers to a spot or "pixel" on the screen that displays a different color than the one assigned to it.

A simple example of artifacts using the ATARI computer is shown by entering the following lines:

```
GRAPHICS  8
COLOR 1
POKE 710,0
PLOT 60,60
PLOT 63,60
```

These statements will plot two points on a black background, however each pixel will have a different color.

To understand the cause of these differing colors one must first understand that all the display information for the TV display is contained in a modulated TV signal.

The two major components of this signal are the luminance, or brightness, and the color, or tint. The luminance information is the primary signal, containing not only the brightness data but also the horizontal and vertical syncs and blanks. The color signal contains the color information and is combined or modulated into the luminance waveform.

The luminance of a pixel on the screen is directly dependent on the amplitude of the luminance signal at that point. The higher the amplitude of the signal, the brighter the pixel.

The color information, however, is a phase shifted signal. A phase shifted signal is a constantly oscillating waveform that has been delayed by some amount of time relative to a reference signal, and this time delay is translated into the color.

The color signal oscillates at a constant rate of about 3.579 MHz, thus defining the highest horizontal color resolution of a TV set. This appears on the screen in the form of 160 visible color cycles across one scan line. (There are actually 228 color cycles including the horizontal blank and sync, and any overscan.)

TELEVISION ARTIFACTS

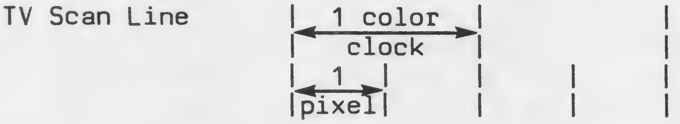
The term "color clock" refers to one color cycle and is the term generally used throughout the ATARI documentation to describe units of measurement across the screen. The graphics mode 7 is an example of one color clock resolution, where each color clock pixel can be a different color. (There are microprocessor limitations though.)

ATARI also offers a "high resolution" mode (GRAPHICS 8) that displays 320 pixels across one line. This is generated by varying the amplitude of the luminance signal at about 7.16 MHz, which is twice the color frequency.

Since the two signals are theoretically independent, one should be able to assign a "background" color to be displayed and then merely vary the luminance on a pixel by pixel basis. This in fact is the way mode 8 works, the "background" color coming from playfield register 2, and the luminances coming from both playfield registers 1 and 2.

The problem is that in practice the color and lumincance signals are not independent. They are part of a modulated signal that must be demodulated to be used. Since the luminance is the primary signal, whenever it changes, it also forces a change in the color phase shift. For one or more color clocks of constant luminance this is no problem, since the color phase shift will be unchanged in this area. However, if the luminance changes on a half color clock boundary it will force a fast color shift at that point. Moreover, that color cannot be altered from the transmitting end of the signal (the ATARI computer).

Since the luminance can change on half color clock boundaries, this implies that two false color, or artifact pixel types can be generated. This is basically true. However, these two pixels can be combined to form two types of full color clock pixels. This is illustrated below:



Luminance 0=off 1=on					
	0	1	0	0	1/2 cc pixel color A
	1	0	0	0	1/2 cc pixel color B
	1	1	0	0	1 cc pixel color C
	0	1	1	0	1 cc pixel color D

Note that each of these pixels requires one color clock of distance and therefore has a horizontal resolution of 160.

The colors A through D are different for each TV set, usually because the tint knob settings vary. Thus they cannot be described as absolute colors, for example red; but they are definitely distinct from each other, and programs have been written that utilize these colors.

To illustrate a simple application of artifacting, refer to the example below. This program draws lines in each of the four artifact colors and then fills in areas using three of the colors. (Note that displaying many pixels of either type C or D next to each other results in the same thing: a line or constant luminance with background color.)

The POKE 87,7 command causes the OS to treat this mode as mode 7 and to use two-bit masks when setting bits in the display memory. To generate color A, use COLOR 1, color B uses COLOR 2, and color C uses COLOR 3. Color D is generated by displaying COLOR 1 to the left of COLOR 2.

```
10    GRAPHICS 8:POKE 87,7:POKE 710,0:POKE 709,14
20    COLOR 1:PLOT 10,5:DRAWTO 10,70
30    PLOT 40,5:DRAWTO 40,70
40    COLOR 2:PLOT 20,5:DRAWTO 20,70
50    PLOT 41,5:DRAWTO 41,70
60    COLOR 3:PLOT 30,5:DRAWTO 30,70
70    FOR X=1 TO 3:COLOR X:POKE 765,X
80    PLOT X*25+60,5:DRAWTO X*25+60,70
90    DRAWTO X*25+40,70:POSITION X*25+40,5
100   XIO 18,#6,12,0,"S:"
110   NEXT X
```

APPENDIX V FLOATING POINT ROUTINE EXAMPLE

```
0000        20           *=   $4000      ;ARBITRARY STARTING POINT
DDB6        30 FMOVE      =       $DDB6
DA60        40 FSUB       =       $DA60
0482        50 FTEMP      =       $0482
DDA7        60 FSTOR      =       $DDA7
D8E6        70 FASC       =       $D8E6
00F3        80 INBUFF     =       $00F3
D800        85 AFP        =       $D800
00F2        90 CIX        =       $00F2
0580        0100 LBUFF     =       $0580
            0110 ;
009B        0120 CR        =       $9B
0009        0130 PUTREC    =       $09
0005        0140 GETREC    =       $05
E456        0150 CIOV      =       $E456
0342        0160 ICCOM     =       $0342
0344        0170 ICBAL     =       $0344
0348        0180 ICBLL    =      $0348
            0190 ; WRITTEN BY CAROL SHAW
            0200 ; FLOATING POINT (F.P.) ROUTINE DEMO PROGRAM.
            0210 ; READS TWO NUMBERS FROM SCREEN EDITOR,
                   CONVERTS THEM TO FLOATING POINT.
            0220 ; SUBTRACTS THE FIRST FROM THE SECOND,
                   STORES THE RESULT IN FTEMP (USER-
            0230 ; DEFINED F.P. REGISTER), AND DISPLAYS THE RESULT.
            0240 ;
            0250 START
4000 205340 0260        JSR  GETNUM    ;GET 1ST NUMBER FROM E:
                                        AND CONVERT TO F.P.
4003 20B6DD 0270        JSR  FMOVE     ;MOVE NUMBER FROM FR0 TO FR1
4006 205340 0280        JSR  GETNUM    ;GET 2ND NUMBER FROM E:
                                        -- OMIT IF ONLY ONE ARGUMENT
4009 2060DA 0290        JSR  FSUB      ;FR0 <-- FR0 - FR1
                                        CHANGE TO GET DIFFERENT ROUTINES
400C 900A   0300        BCC  NOERR     ;SKIP IF NO ERROR
            0310 ;
            0320 ; ERROR -- DISPLAY MESSAGE
            0330 ;
400E A981   0340        LDA  #ERRMSG&255
4010 8D4403 0350        STA  ICBAL
4013 A940   0360        LDA  #ERRMSG/256
4015 4C3940 0370        JMP  CONTIN
            0380 NOERR
4018 A282   0390        LDX  #FTEMP&255       ;STORE RESULT IN
                                               FTEMP (USER'S F.P. VAR)
401A A004   0400        LDY  #FTEMP/256
401C 20A7DD 0410        JSR  FSTOR
```

```
            0420 ;
            0430 ; CONVERT NUMBER TO ASCII STRING.
            0440 ; FIND END OF STRING AND CHANGE NEGATIVE # TO
                   POSITIVE AND ADD CARRIAGE RETURN.
            0450 ;
401F 20E6D8 0460          JSR  FASC       ;CONVERT FROM F.P. TO
                                           ASCII STRING IN LBUFF
4022 A0FF   0470          LDY  #$FF
            0480 MLOOP
4024 C8     0490          INY
4025 B1F3   0500          LDA  (INBUFF),Y ;LOAD NEXT BYTE
                                            (POINTED TO BY INBUFF). POSITIVE?
4027 10FB   0510          BPL  MLOOP      ;YES. CONTINUE
4029 297F   0520          AND  #$7F       ;NO. NEGATIVE -- MASK OFF MSBIT
402B 91F3   0530          STA  (INBUFF),Y
402D C8     0540          INY
402E A99B   0550          LDA  #CR        ;STORE CARRIAGE RETURN
4030 91F3   0560          STA  (INBUFF),Y
            0570 ;
            0580 ; DISPLAY RESULT
            0590 ;
4032 A5F3   0600          LDA  INBUFF     ;BUFFER ADDRESS IS IN INBUFF
4034 8D4403 0610          STA  ICBAL
4037 A5F4   0620          LDA  INBUFF+1
            0630 CONTIN
4039 8D4503 0640          STA  ICBAL+1
403C A909   0650          LDA  #PUTREC    ;COMMAND IS PUT RECORD
403E 8D4203 066U          STA  ICCOM
4041 A928   0670          LDA  #40        ;BUFFER LENGTH = 40
4043 8D4803 0680          STA  ICBLL
4046 A900   0690          LDA  #0
4048 8D4903 0700          STA  ICBLL+1
404B A200   0710          LDX  #0         ;IOCB # = 0 (SCREEN EDITOR)
404D 2056E4 0720          JSR  CIOV       ;CALL CIO
4050 4C0040 0730          JMP  START      ;DO IT AGAIN
```

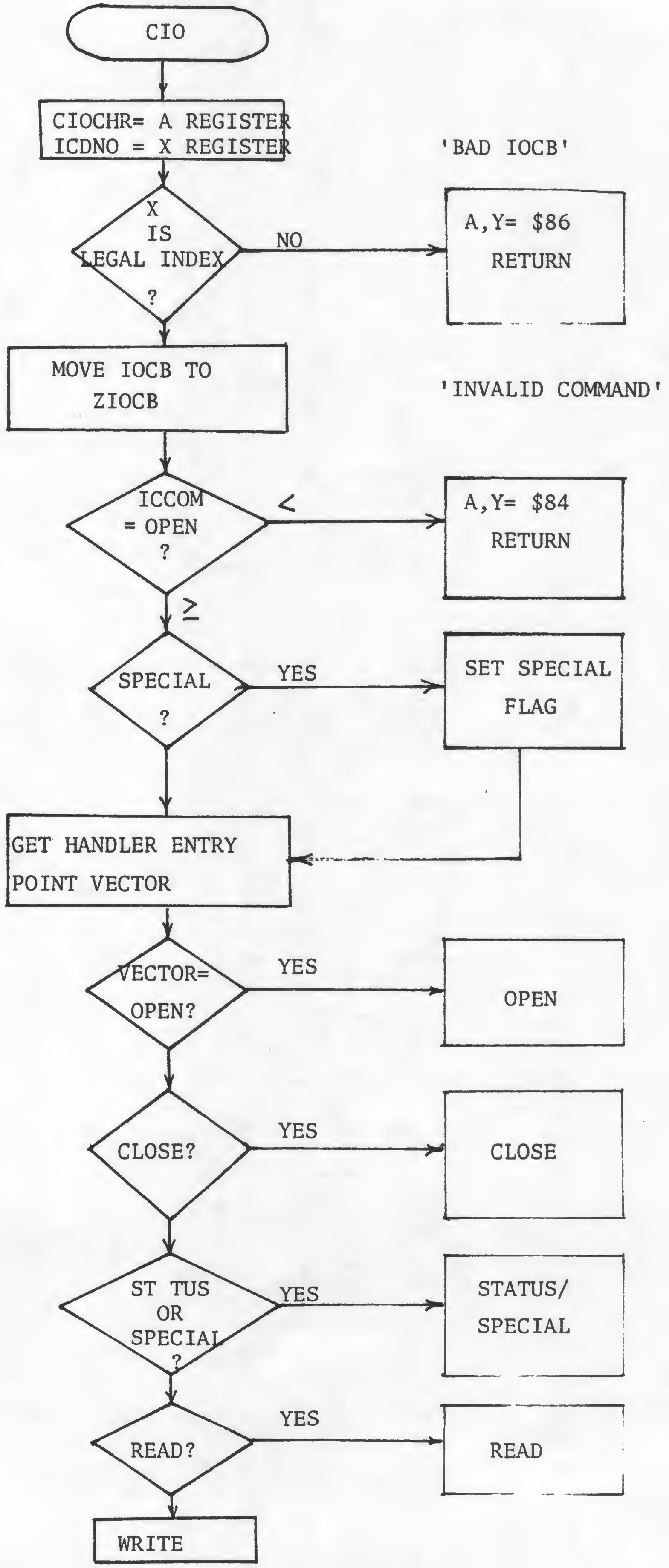
```
             0740 ;
             0750 ; GETNUM -- GET ASCII STRING FROM E: AND
                    CONVERT TO F.P. IN FR0
             0760 ;
             0770 GETNUM
4053 A905    0780        LDA  #GETREC   ;GET RECORD (ENDS IN CR)
4055 8D4203  0790        STA  ICCOM
4058 A980    0800        LDA  #LBUFF&255   ;BUFFER ADDRESS = LBUFF
405A 8D4403  0810        STA  ICBAL
405D A905    0820        LDA  #LBUFF/256
405F 8D4503  0830        STA  ICBAL+1
4062 A928    0840        LDA  #40       ;BUFFER LENGTH = 0
4064 8D4803  0850        STA  ICBLL
4067 A900    0860        LDA  #0
4069 8D4903  0870        STA  ICBLL+1
406C A200    0880        LDX  #0        ;IOCB # = 0 (SCREEN EDITOR)
406E 2056E4  0890        JSR  CIOV      ;CALL CIO
4071 A980    0900        LDA  #LBUFF&255     ;STORE BUFFER ADDRESS
                                              IN POINTER (INBUFF)
4073 85F3    0910        STA  INBUFF
4075 A905    0920        LDA  #LBUFF/256
4077 85F4    0930        STA  INBUFF+1
4079 A900    0940        LDA  #0        ;BUFFER INDEX = 0
407B 85F2    0950        STA  CIX
407D 4C00D8  0960        JMP  AFP       ;CALL ASCII TO FLOATING POINT AND
RETURN
4080 60      0970 INIT   RTS            ;POWER UP ROUTINE (DO NOTHING)
4081 45      0980 ERRMSG .BYTE "ERROR",CR ;INDICATES CARRY SET
                                           ON RETURN FROM FP ROUTINE
4082 52
4083 52
4084 4F
4085 52
4086 9B
             0990 ;
             1000 ; ROUTINE START INFO
             1010 ;
4087         1020        *=   $2E0
02E0 0040    1030        .WORD   START
02E2         1040        .END
```

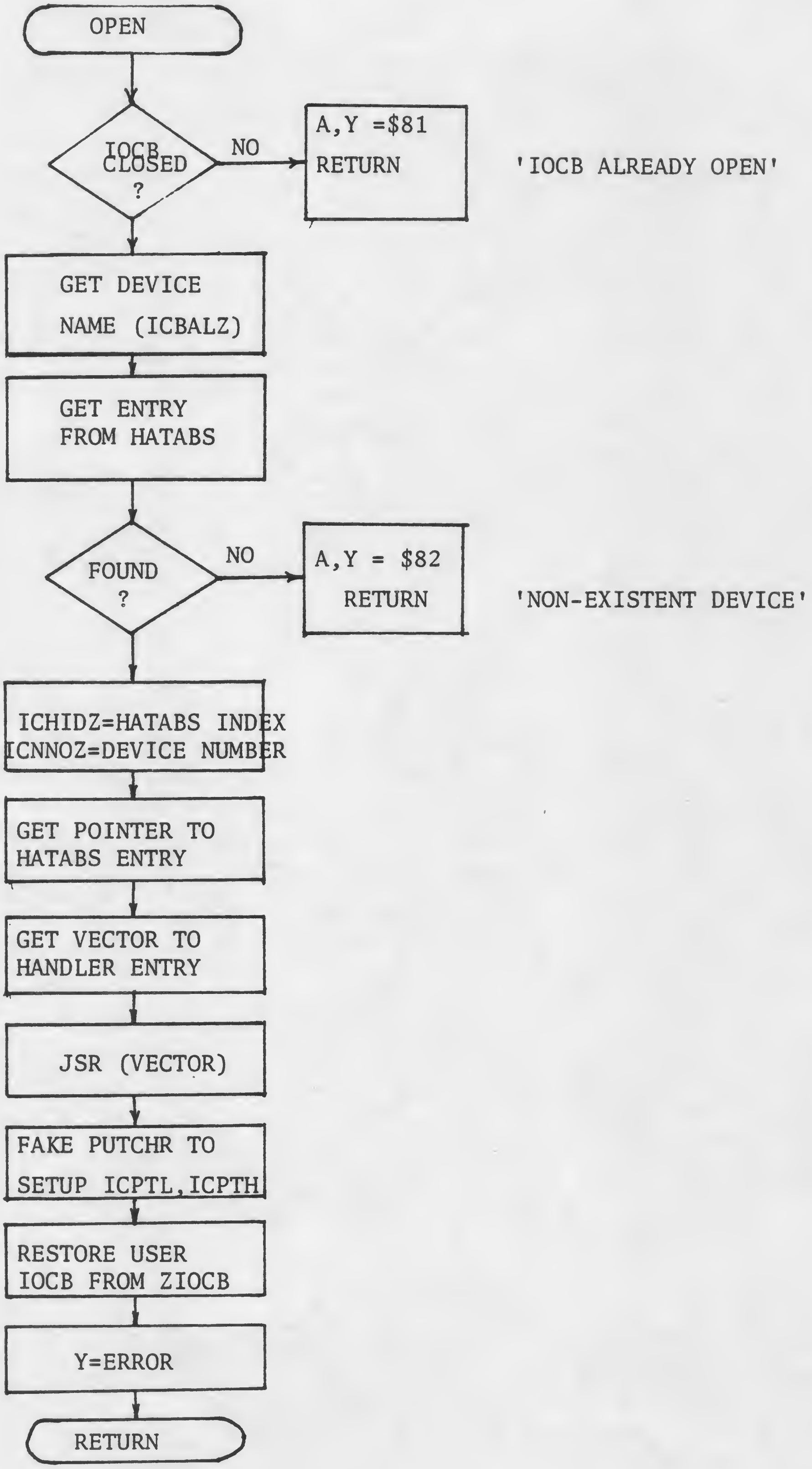
FLOATING POINT ROUTINES

NAME	ADDRESS	DESCRIPTION	APPROXIMATE MAXI-TIME(usec)
AFP	D800	Ascii to Floating Point	3500
FASC	D8E6	Floating Point to Ascii	950
IFP	D9AA	Integer to Floating Point	1330
FPI	D9D2	Floating Point to Integer	2400
FSUB	DA60	FR0 -- FR0 - FR1 Subtraction	740
FADD	DA66	FR0 -- FR0 + FR1 Addition	710
FMUL	DADB	FR0 -- FR0 * FR1 Multiplication	12000
FDIV	DB28	FR0 -- FR0 / FR1 Division	10000
FLD0R	DD89	Floating Load FR0 using X,Y	70
FLD0P	DD8D	Floating Load FR0 using FLPTR	60
FLD1R	DD98	Floating Load FR1 using X,Y	70
FLD1P	DD9C	Floating Load FR1 using FLPTR	60
FSTOR	DDA7	Floating Store FR0 using X,Y	70
FSTOP	DDAB	Floating Store FR0 using FLPTR	70
FMOVE	DDB6	FR0 -- FR1 F.P. Move	60
PLYEVL	DD40	Polynomial Evaluation	88300
EXP	DDC0	FR0 -- e^{FR0} exponentiation	115900 ≈.1s
EXP10	DDCC	FR0 -- 10^{FR0} exponentiation	108800
LOG	DECD	FR0 -- LOG_e(FR0) natural log	136000
LOG10	DED1	FR0 -- LOG_{10}(FR0) common log	125400
ZFR0	DA44	FR0 -- 0	80
AF1	DA46	clear page zero F.P. reg. (6 bytes)	80
In BASIC Cartridge:			
SIN	BDA7	FR0 -- SIN(FR0)	79400
COS	BDB1	FR0 -- COS(FR0)	77400
ATAN	BE77	FR0 -- ATAN(FR0)	126700
SQR	BEE5	FR0 -- SQUARE ROOT(FR0)	131100

Times are for worst case, including JSR and RTS.
Times are approximate.
1 sec. = 1000000 usec.
Times are approximately 30%-40% less with DMA disabled (SDMCR):
POKE 559,0 disable DMA
POKE 559,34 enable DMA

APPENDIX VI - CIO




OPEN
IOCB CLOSED ?
NO
A,Y =$81
RETURN
'IOCB ALREADY OPEN'
GET DEVICE NAME (ICBALZ)
GET ENTRY FROM HATABS
FOUND ?
NO
A,Y = $82
RETURN
'NON-EXISTENT DEVICE'
ICHIDZ=HATABS INDEX
ICNNOZ=DEVICE NUMBER
GET POINTER TO HATABS ENTRY
GET VECTOR TO HANDLER ENTRY
JSR (VECTOR)
FAKE PUTCHR TO SETUP ICPTL,ICPTH
RESTORE USER IOCB FROM ZIOCB
Y=ERROR
RETURN

APPENDIX VII - QUME PRINTER HANDLER

```
1 ;HANDLER USES THE FRONT PORTS TO SEND DATA TO A QUME PRINTER
 10  *=$3300
20 CR=$9B
30 SPCE=$20
40 PIAB=$D301
50 PIAC=$D303
60 DOSVEC=$0A
70 DOSINI=$0C
80 HATABS=$031A
90 MEMLO=$2E7
0100 ;
0110 ; HANDLER ENTRY TABLE
0120 ;
0130 QHTBL .WORD QOPEN-1
0140  .WORD QXIT-1 "CLOSE"
0150  .WORD QERR-1 "GET"
0160  .WORD QPUT-1
0170  .WORD QXIT-1 "STATUS"
0180  .WORD QXIT-1 "SPECIAL"
0190  JMP QOPEN
0200 QHOOK1 LDA         DOSINI INIT ON LOAD
0210  STA DOSLNK
0220  LDA DOSINI+1
0230  STA DOSLNK+1
0240 QH000 JSR QINST    TRY TO INSTALL
0250  BCC QH001         BY STEALING DOSINI
0260  RTS PASS          RETURN
0270 QH001 LDA #QHOOK&255
0310  STA DOSINI
0320  LDA #QHOOK/256
0360  STA DOSINI+1
0370  RTS               RETURN TO DOS
0380 QHOOK2 JSR JIND    FIRST INIT DOS
0390  JMP QH000         THEN INSTALL US
0400 JIND JMP (DOSLNK)  INIT DOS TOO
0410 DOSLNK .WORD $E477 LNK FOR INIT
0420 PITCH .BYTE $80
0430 VPTCH .BYTE $60
0440 ACUH .WORD 0
0450 SPACES .WORD 0
0460 QWD .WORD 0
```

```
0470 ;
0480 ; INIT QUME AND
0490 ; SOFTWARE VARIABLES
0500 ;
0510 QOPEN LDA #0
0520  LDX #5
0530 QILP STA ACUH,X    CLR VARS
0540  DEX
0550  BPL QILP
0560 QSET LDA PIAC      SET FRONT PORTS
0570  AND #$FB
0580  STA PIAC          PT TO DDR
0590  LDY #7
0600  STY PIAB          SET 3 OUT
0610  ORA #4
0620  STA PIAC
0630  LDA #2
0640  STA PIAB
0650  BNE QSNDU RESTORE
0660 ;
0670 ; SEND A CONTROL WORD TO QUME
0680 ;
0690 SNDYA STY QWD-1
0700 SNDA STA QWD
0710 QSND LDA PIAB
0720  AND #8
0730  BNE QSND          WAIT FOR RDY
0740 QSNDU LDY #16
0750  LDA QWD
0760  EOR #7
0770  STA QWD
0780 QSLP LDA QWD
0790  AND #1
0800  ORA #2
0810  STA PIAB
0820  JSR QDELAY
0830  AND #1
0840  STA PIAB
0850  JSR QDELAY
0860  ORA #2
0870  STA PIAB
0880  LSR QWD+1
0890  ROR QWD
0900  DEY
0910  BNE QSLP
0920  JSR QDELAY
0930  ORA #4 STROBE
0940  STA PIAB
0950  JSR QDELAY
0960  AND #3
0970  STA PIAB
0980  RTS
```

```
0990 ;
1000 ; DELAY FOR LINE SETTLING
1010 ;
1020 QDELAY LDX #60
1030 QDLPO DEX
1040  BNE QDLPO
1050  RTS
1060 ;
1070 ; SEND A CHARACTER FROM ACC
1080 ;
1090 QPUT CMP #CR
1100  BEQ QCR
1110  CMP #SPCE
1120  BCC RLY1
1130  BNE PUTPRT
1140  INC SPACES
1150 RLY1 BNE RLY2       (ALWAYS)
1160 QCR LDA ACUH+1
1170  ORA #$40           (NEGATE IT)
1180  TAY
1190  LDA ACUH
1200  ORA #2
1210  JSR SNDYA          SEND TO QUME
1220  LDA VPTCH          VERTICAL PITCH
1230  ORA #3
1240  JSR SNDA           DO LINE FEED
1250  LDA #0
1260  STA ACUH
1270  STA ACUH+1
1280  STA SPACES
1290 RLY2 JMP QXIT
1300 PUTPRT PHA          SAVE CHAR
1310  LDX SPACES
1320  BEQ PPX
1330  LDA #0
1340  TAY
1350 PPLP CLC
1360  ADC PITCH
1370  BCC PPO
1380  INY
```

```
1390 PPO DEX
1400  BNE PPLP
1410  PHA
1420  STX SPACES
1430  CLC
1440  ADC ACUH
1450  STA ACUH
1460  TYA
1470  ADC ACUH+1
1480  STA ACUH+1
1490  PLA
1500  ORA #2
1510  JSR SNDYA
1520 PPX LDA #0
1530  STA QWD+1
1540  PLA GET CHAR BACK
1550  ASL A
1560  BCC NOUND
1570  LDY #5
1580  PHA
1590  LDA #$F1
1600  JSR SNDYA           UNDERLINE IT INVERSE
1610  PLA
1620  CMP #SPCE+SPCE
1630  CLC
1640  BNE NOUND          NOT A SPACE
1650  INC SPACES
1660  JMP QXIT
1670 NOUND ROL QWD+1
1680  ASL A
1690  ROL QWD+1
1700  ASL A
1710  ROL QWD+1
1720  ASL A
1730  ROL QWD+1
1740  ORA #1
1750  INC SPACES
1760  JSR SNDA
1770 QXIT LDY #1
1780  RTS
```

```
1790 ;
1800 ; ERROR EXIT
1810 ;
1820 QERR LDY #$8B
1830  RTS
1840 ;
1850 ; INSTALL HANDLER IN HATABS
1860 ;
1870 QINST LDY #0
1880 QINLP LDA HATABS,Y
1890  CMP #$50 FIND P:
1900  BEQ QIPUT
1910  INY
1920  INY
1930  INY
1940  CPY #$21           END OF TABLE?
1950  BCC QINLP          NO, LOOP
1960  RTS                ELSE RETURN
1970 QIPUT LDA #QHTBL&255
2010  STA HATABS+1,Y
2020  LDA #QHTBL/256
2060  STA HATABS+2,Y
2070  LDA #QEND&255
2110  STA MEMLO          SET MEMLOL
2120  LDA #QEND/256
2160  STA MEMLO+1        SET MEMLOH
2170  CLC
2180  RTS
2190 QEND=*
2200  .END
```

RANDOM ACCESS
by Bill Bartlett

DEFINITION OF RANDOM ACCESS

Random access is defined as any method of reading or writing records from/to any part of a data file without first having to read through previous records of the file.

Before any I/O command can be processed, the storage media must be physically positioned on the device to the correct byte location. A sequential device such as the ATARI 410 cassette tape drive must be positioned manually by the operator using the tape counter. Thus, if record 100 is desired, the tape drive must first bypass records 1-99. It should be obvious that processing is very time consuming simply because of the physical nature of the device.

A random device such as the ATARI 810 disk drive can be positioned to any byte in a file under the control of a BASIC porgram. Thus if record 100 is desired, the disk drive can position there immediately. Random access is accomplished in a BASIC program with the POINT command.

```
OPEN #1,12,0,"D1:--------"
SECTOR=63
BYTE=26
POINT #1,SECTOR,BYTE
```

The above commands cause the file opened on channel 1 to be positioned to sector 63, byte 26. The file must have been previously opened in mode 12 and sector 63 must have been allocated to the file by the File Manangement System (FMS).

GOALS

1. Define a data file structure that facilitates random access.

2. Maximize diskette utilization by using all available user space in a file.

3. Simplify coding by developing subroutines to handle the most common random access processing.

CONCEPTS

1. Random access can be facilitated in 2 ways.

First, store the sector and byte information needed by the POINT command in the data file so that it can be readily assigned to a program variable when the file is opened. Second, have the program variable use as little RAM as possible.

A two-dimensional numeric array could be used as the program variable. However, each record would require 12 bytes of RAM. A string variable will use less RAM because the sector number ranging 0-720 can be stored in 2 bytes and the byte offset ranging 0-127 can be stored in 1 byte.

2. Maximum diskette utilization is accomplished by using mode 12 (I/O) instead of mode 9 (append) in maintenance programs.

There are 2 problems with using mode 9 to access a data file. First, when the OPEN command is processed, the FMS will always allocate a new sector to the file regardless of whether the last sector is completely filled with data. Second, the POINT command will not execute on a file opened in mode 9.

Mode 12 is facilitated by allocating blank records to the data file when it is created.

3. Record allocation is facilitated by having a 1 byte status byte for each record. A value of 0 indicates the record is not active. A value of 1 indicates the record is active.

FILE STRUCTURE

A data file will consist of 3 sections.

FILE HEADER RECORD
FILE RANDOM ACCESS POINTERS
DATA RECORDS

The file header record is the first sector of the file. It is 125 bytes long (124 data + 1 delimiter).

FILE HEADER RECORD

BYTE	CONTENTS
1-2	sector address of file header record
3	byte offset of file header record (0)
4	not used
5-6	# records in file
7-8	# bytes in record
9-124	not used

The file random access pointers immediately follow the file header record and thus start in the second sector of the file. The data is stored as a string variable consisting of groups of 4 bytes. The first 4 bytes are used to loacte the pointer strings own location in the file. Then there are 4 bytes for each record in the file.

FILE RANDOM ACCESS POINTERS

BYTE	CONTENTS
1-2	sector address of file random access pointers
3	byte offset of file random access pointers
4	not used

```
5-N   4 bytes for each record
      1-2  sector address of record
      3    byte offset of record
      4    status of record
```

The data records immediately follow the file random access pointers. They are stored as string variables

RANDOM ACCESS SUBROUTINES

1. FILEOPEN.SUB

This routine opens a file in mode 12 and initializes the random access variables used in the other routines.

```
     Input variables:
          FILE$     - must be dimensioned and assigned by the user (15 bytes)
          CHANNEL   - IOCB number (1-5)

     Call:
          GOSUB 9300

     Output variables:
          FILEMAX   - # records in file
          FILELEN   - # bytes in record
          FILEPTR$ - contains the random access pointers
          FILEREC$ - dimensioned for record I/O

     Scratch variables:
          FILESEC
          FILEBYT     These are not currently used
          FILESTS
```

2. FILEADD.SUB

This routine allocates the next available record by reading through FILEPTR$ looking for a status byte of 0. When a status of 0 is found, it is set to 1 and the record# is stored in RECORD. If RECORD returns a value of 0, all records in the file are active.

```
     Input variables:
          none

     Call:
          GOSUB 9400

     Output variables:
          RECORD    - record# of next available record
          FILEPTR$ - updated with status byte of 1
```

Scratch variables:
RECORD1
B

3. FILEDEL.SUB

This routine flags a record inactive by changing its status byte in FILEPTR$ to 0.

Input variables:
RECORD

Call:
GOSUB 9450

Output variables:
FILEPTR$ - updated with status byte of 0

Scratch variables:
B

4. FILEPTR.SUB

This routine updates the random access pointer section of the data file with the current value of FILEPTR$.

Input variables:
none

Call:
GOSUB 9500

Output variables:
none

Scratch variables:
S
B

5. FILEPOS.SUB

This routine positions the file pointer to the beginning of a record.

Input variables:
RECORD - record# to point to.

Call:
GOSUB 9600

```
Output variables:
     STS      - value of status byte for record

Scratch variables:
     S
     B
```

```
9200 REM APPENDIX AND ROUTINES BY WILLIAM BARTLETT
9300 REM 'FILEOPEN.SUB' WBB 3-30-81
9305 REM OPEN A FILE IN MODE 12 AND DEFINE ALL VARIABLES
9310 OPEN #CHANNEL,12,0,FILE$
9315 DIM FILEHED$(124)
9320 FOR I=1 TO 124:GET #CHANNEL,B:FILEHED$(I)=CHR$(B):NEXT I:GET #CHANNEL,B
9325 FILESEC=ASC(FILEHED$(1))*256+ASC(FILEHED$(2))
9330 FILEBYT=ASC(FILEHED$(3))
9335 FILESTS=ASC(FILEHED$(4))
9340 FILEMAX=ASC(FILEHED$(5))*256+ASC(FILEHED$(6))
9345 FILELEN=ASC(FILEHED$(7))*256+ASC(FILEHED$(8))
9350 DIM FILEPTR$(4+4*FILEMAX),FILEREC$(FILELEN)
9355 FOR I=1 TO 4+4*FILEMAX:GET #CHANNEL,B:FILEPTR$(I)=CHR$(B):NEXT I:GET
#CHANNEL,B
9360 RETURN
```

```
9400 REM 'FILEADD.SUB' WBB 3-30-81
9405 REM ALLOCATE THE NEXT AVAILABLE RECORD
9410 RECORD=0
9415 IF FILEMAX=0 THEN RETURN
9420 FOR RECORD1=1 TO FILEMAX
9425 B=RECORD1*4+4
9430 IF FILEPTR$(B,B)=CHR$(0) THEN
RECORD=RECORD1:RECORD1=FILEMAX:FILEPTR$(B,B)=CHR$(1)
9435 NEXT RECORD1
9440 RETURN
```

```
9450 REM 'FILEDEL.SUB' WBB 3-30-81
9455 REM DELETE AN ACTIVE RECORD
9460 B=RECORD*4+4
9465 FILEPTR$(B,B)=CHR$(0)
9470 RETURN
```

```
9500 REM 'FILEPTR.SUB' WBB 3-19-81
9510 REM WRITE FILEPTR$
9515 S=ASC(FILEPTR$(1))*256+ASC(FILEPTR$(2))
9520 B=ASC(FILEPTR$(3))
9525 POINT #CHANNEL,S,B
9530 FOR I=1 TO 4+4*FILEMAX:B=ASC(FILEPTR$(I)):PUT #CHANNEL,B:NEXT I
9535 RETURN
```

```
9600 REM 'FILEPOS.SUB' WBB 3-31-81
9605 REM POINT FILE TO RECORD
9610 S=ASC(FILEPTR$(RECORD*4+1))*256+ASC(FILEPTR$(RECORD*4+2))
9615 B=ASC(FILEPTR$(RECORD*4+3))
9620 STS=ASC(FILEPTR$(RECORD*4+4))
9625 POINT #CHANNEL,S,B
9630 RETURN
```

I. INTRODUCTION TO FILE001

This program is used to create a new disk data file, allocate its file space, and initialize its random access structure. The random access structure is designed to interface with the following BASIC subroutines.

FILEOPEN.SUB
FILEADD.SUB
FILEDEL.SUB
FILEPTR.SUB
FILEPOS.SUB

II. PROGRAM STRUCTURE

0001-0999 main line logic
1000-9999 subroutines

III. PROGRAM LOGIC The 5 major execution phases are program initialization, file definition, screen setup, file allocation, and closing.

A. Initialization 1000-1495

1015	dimension variables
1020	identify program to user
1025-1035	allow user to exit program
1040	set up BLANK$ to be used as a string filler

B. File Definition 1500-1999

1510-1535	have user define parameters of file
1540-1550	allow user to redefine parameters until correct
1600-1645	search diskette dircetory to insure that the file specified doesn't already exist

```
     1700-1730 verify there are enough free sectors to create the file
     1800-1825 concatenate the filename
```

C. Screen setup 9600-9650

D. File Allocation 2000-2499

```
     2005      create the file
     2010      save pointers
     2015      dimension variables
               FILEHED$ - file header record
               FILEPTR$ - file pointer variable
               FILEREC$ - record I/O variable
     2100-2180 set up FILEHED$ and send to file
     2200-2245 set up FILEPTR$ and send to file
     2305      blank out FILEREC$
     2310-2345 store record pointers in FILEPTR$
     2350      refresh screen
     2355      send FILEREC$ to file
     2370-2375 send final FILEPTR$ to file
```

E. Closing 0900-0999

```
10 REM 'FILE0001'  WBB 3-12-81
100 REM MAIN LINE
110 GOSUB 1000
120 IF YN$="N" THEN 900
130 GOSUB 1500
140 GOSUB 9600
150 GOSUB 2000
900 REM END
910 CLOSE #1
920 GRAPHICS 0
930 END
1000 REM INIT
1005 TRAP 9800
1010 GRAPHICS 2
1015 DIM YN$(1),DVC$(3),FILE$(8),EXT$(3),FILENAME$(15),FMS$(16),BLANK$(128)
1020 PLOT 5,4:PRINT #6;"FILE0001"
1025 PRINT "THIS PROGRAM INITIALIZES A NEW FILE."
1030 PRINT "DO YOU WISH TO PROCEED (Y/N) ";
1035 INPUT YN$
1040 BLANK$=" ":BLANK$(128)=" ":BLANK$(2)=BLANK$
1095 RETURN
1500 REM DEF FILE
1505 GRAPHICS 2
1510 PRINT #6;"FILE DEFINITION"
1515 PRINT "    DEVICE ";:INPUT DVC$:DVC$(LEN(DVC$)+1)=":"
1520 PRINT " FILE NAME ";:INPUT FILE$
1525 PRINT " EXTENTION ";:INPUT EXT$
```

```
1530 PRINT " # RECORDS ";:INPUT FILEMAX
1535 PRINT "REC LENGTH ";:INPUT FILELEN
1540 PRINT "DO YOU WISH TO PROCEED (Y/N) ";
1545 INPUT YN$
1550 IF YN$<>"Y" THEN 1500
1600 REM VERIFY FILE DOESNT EXIST
1605 FMS$=DVC$:FMS$(LEN(FMS$)+1)="*.*"
1610 OPEN #1,6,0,FMS$
1615 INPUT #1,FMS$
1620 IF FMS$(2,2)<>" " THEN 1700
1625 IF FMS$(3,2+LEN(FILE$))<>FILE$ OR FMS$(11,10+LEN(EXT$))<>EXT$ THEN 1615
1630 CLOSE #1
1635 PRINT "FILE ALREADY EXISTS!";CHR$(253)
1640 GOSUB 9850
1645 GOTO 1500
1700 REM VERIFY ENOUGH DISK SPACE EXISTS
1705 CLOSE #1
1710 FREE=VAL(FMS$(1,3)):NEED=FILEMAX*(FILELEN+5)/125+1
1715 IF NEED<FREE THEN 1800
1720 PRINT "INSUFF FILE SP! ":PRINT FREE;" FREE   ";NEED;" NEEDED!";CHR$(253)
1725 GOSUB 9850
1730 GOTO 1500
1800 REM CONCATENATE FILENAME
1805 FILENAME$=DVC$
1810 FILENAME$(LEN(FILENAME$)+1)=FILE$
1815 FILENAME$(LEN(FILENAME$)+1)="."
1820 FILENAME$(LEN(FILENAME$)+1)=EXT$
1825 RETURN
2000 REM INIT HEADER,POINTER,RECORD STRINGS
2005 OPEN #1,8,0,FILENAME$
2010 NOTE #1,FILESEC,FILEBYT
2015 DIM FILEHED$(124),FILEPTR$(4+4*FILEMAX),FILEREC$(FILELEN)
2100 REM FILE HEADER
2105 FILEHED$=BLANK$
2110 HI=INT(FILESEC/256)
2115 LO=FILESEC-HI*256
2120 FILEHED$(1,1)=CHR$(HI)
2125 FILEHED$(2,2)=CHR$(LO)
2130 FILEHED$(3,3)=CHR$(FILEBYT)
2135 FILEHED$(4,4)=CHR$(0)
2140 HI=INT(FILEMAX/256)
2145 LO=FILEMAX-HI*256
2150 FILEHED$(5,5)=CHR$(HI)
2155 FILEHED$(6,6)=CHR$(LO)
2160 HI=INT(FILELEN/256)
2165 LO=FILELEN-HI*256
2170 FILEHED$(7,7)=CHR$(HI)
2175 FILEHED$(8,8)=CHR$(LO)
2180 PRINT #1;FILEHED$
2200 REM FILE POINTER
2205 FOR I=1 TO 4+4*FILEMAX STEP 128:FILEPTR$(I)=BLANK$:NEXT I
2210 NOTE #1,S,B
```

```
2215 HI=INT(S/256)
2220 LO=S-HI*256
2225 FILEPTR$(1,1)=CHR$(HI)
2230 FILEPTR$(2,2)=CHR$(LO)
2235 FILEPTR$(3,3)=CHR$(B)
2240 FILEPTR$(4,4)=CHR$(0)
2245 PRINT #1;FILEPTR$
2300 REM RECORDS
2305 FOR I=1 TO FILELEN STEP 128:FILEREC$(I)=BLANK$:NEXT I
2310 FOR I=1 TO FILEMAX
2315 NOTE #1,S,B
2320 HI=INT(S/256)
2325 LO=S-HI*256
2330 FILEPTR$(I*4+1,I*4+1)=CHR$(HI)
2335 FILEPTR$(I*4+2,I*4+2)=CHR$(LO)
2340 FILEPTR$(I*4+3,I*4+3)=CHR$(B)
2345 FILEPTR$(I*4+4,I*4+4)=CHR$(0)
2350 GOSUB 9700
2355 PRINT #1;FILEREC$
2360 NEXT I
2365 CLOSE #1
2370 OPEN #1,12,0,FILENAME$
2375 GOSUB 9510
2380 RETURN
```

```
9500 REM 'FILEPTR.SUB' WBB 3-19-81
9510 REM WRITE FILEPTR
9515 S=ASC(FILEPTR$(1,1))*256+ASC(FILEPTR$(2,2))
9520 B=ASC(FILEPTR$(3,3))
9525 POINT #1,S,B
9530 PRINT #1;FILEPTR$
9535 RETURN
9600 REM DISPLAY SCREEN TEMPLATE
9605 GRAPHICS 2
9610 PRINT #6;"* INITIALIZING *"
9615 PRINT #6
9620 PRINT #6;"CURRENT"
9625 PRINT #6;"  TOTAL"
9630 PRINT #6;" % COMP"
9635 PRINT #6
9640 PRINT #6;" SECTOR"
9645 PRINT #6;"   BYTE"
9650 RETURN
9700 REM REFRESH SCREEN
9705 PLOT 10,2:PRINT #6;I
9710 PLOT 10,3:PRINT #6;FILEMAX
9715 PLOT 10,4:PRINT #6;INT(I/FILEMAX*100)
9720 PLOT 10,6:PRINT #6;S;"   "
9725 PLOT 10,7:PRINT #6;B;"   "
9795 RETURN
9800 REM TRAP.SUB
9805 TRAP 9825
9810 PRINT "ERROR ";PEEK(195);" AT ";PEEK(187)*256+PEEK(186)
9815 PRINT "ACKNOWLEDGE ";
9820 INPUT YN$
9825 END
9850 REM 'DELAY.SUB' WBB 3-19-81
9851 REM DELAYS EXECUTION FOR 2.5 SEC
9852 REM (SCRATCH-P20)
9860 P20=PEEK(20)+150:IF P20>255 THEN P20=P20-256
9865 IF PEEK(20)<>P20 THEN 9865
9870 RETURN
```

RANDOM ACCESS

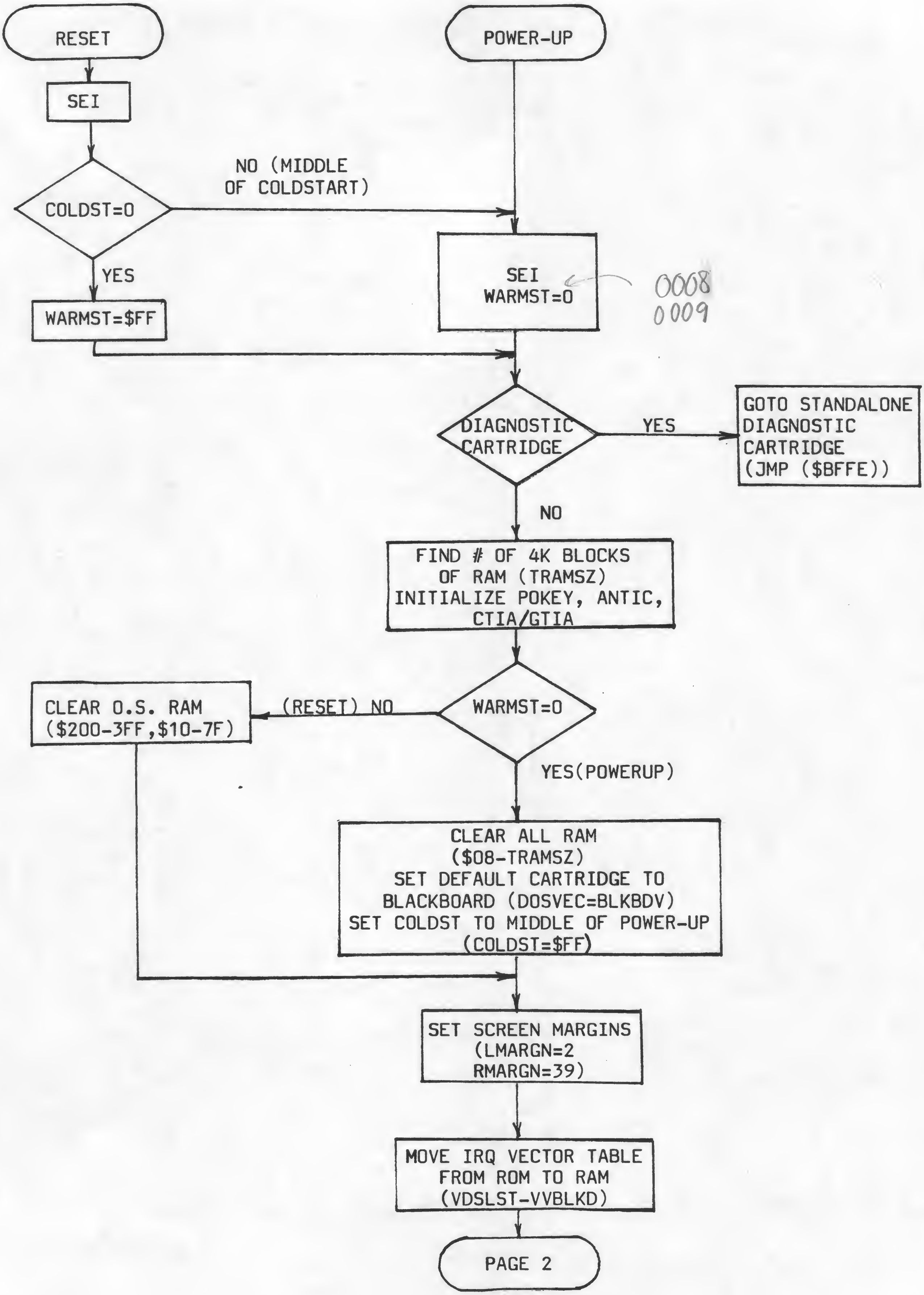
EXAMPLE PROGRAM

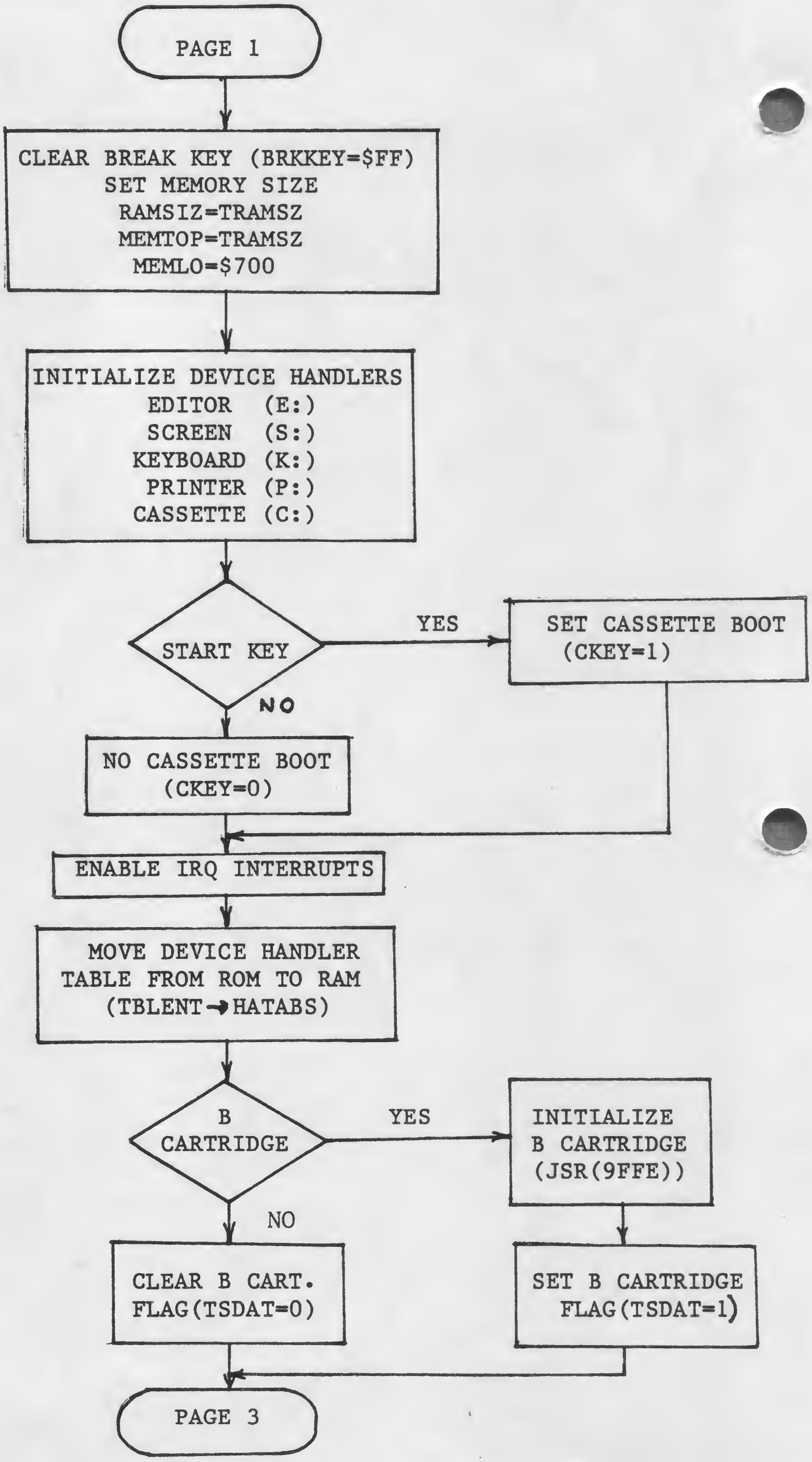
```
10 REM 'FILEEX' WBB 3-31-81
100 REM EXAMPLE OF RANDOM ACCESS ROUTINES
101 REM THIS PROGRAM WILL PROCESS THE FILE D2:AREACODE.DAT
102 REM WHICH SHOULD HAVE BEEN INITIALIZED USING 'FILE0001'
103 REM IT CONSISTS OF 24 BYTE RECORDS: 1-3 AREA CODE, 4-24 LOCATION DESC
104 REM
110 GRAPHICS 0
120 PRINT "'FILEEX'":PRINT :PRINT "INITIALIZING"
200 REM INITIALIZE VARIABLES
210 DIM FILE$(15),ACODE$(3),LOC$(21),YN$(1)
220 CHANNEL=1
230 FILE$="D1:AREACODE.DAT"
300 REM OPEN DATA FILE
310 PRINT "OPENING DATA FILE"
320 GOSUB 9300
400 REM BEGIN OPERATOR INPUT
410 PRINT
420 PRINT "(0=END) AREA CODE ";:INPUT ACODE$:IF ACODE$="0" THEN 900
500 REM SEARCH ACTIVE RECORDS FOR MATCHING AREA CODE
510 MATCH=0
520 FOR RECORD=1 TO FILEMAX
530 GOSUB 9600
540 IF STS=1 THEN GOSUB 5000
550 NEXT RECORD
560 IF MATCH=1 THEN 400
600 REM MATCH NOT FOUND, ALLOW ADD
610 PRINT "MATCH NOT FOUND IN FILE!"
620 PRINT "(Y/N) DO YOU WISH TO ADD ";:INPUT YN$
630 IF YN$<>"Y" THEN 400
700 REM ADD REQUESTED BY OPERATOR
710 GOSUB 9400
720 IF RECORD=0 THEN PRINT "FILE IS FULL, RECORD NOT ADDED!":GOTO 400
730 PRINT "LOCATION: ";:INPUT LOC$
740 FILEREC$=ACODE$
750 FILEREC$(4)=LOC$
800 REM PERFORM FILE UPDATING
810 GOSUB 9600
820 PRINT #CHANNEL;FILEREC$
830 GOSUB 9500
840 GOTO 400
900 REM ALLOW DELETES
910 PRINT
920 PRINT "(Y/N) DO YOU WISH TO DELETE ANY ";:INPUT YN$
930 IF YN$<>"Y" THEN 1200
```

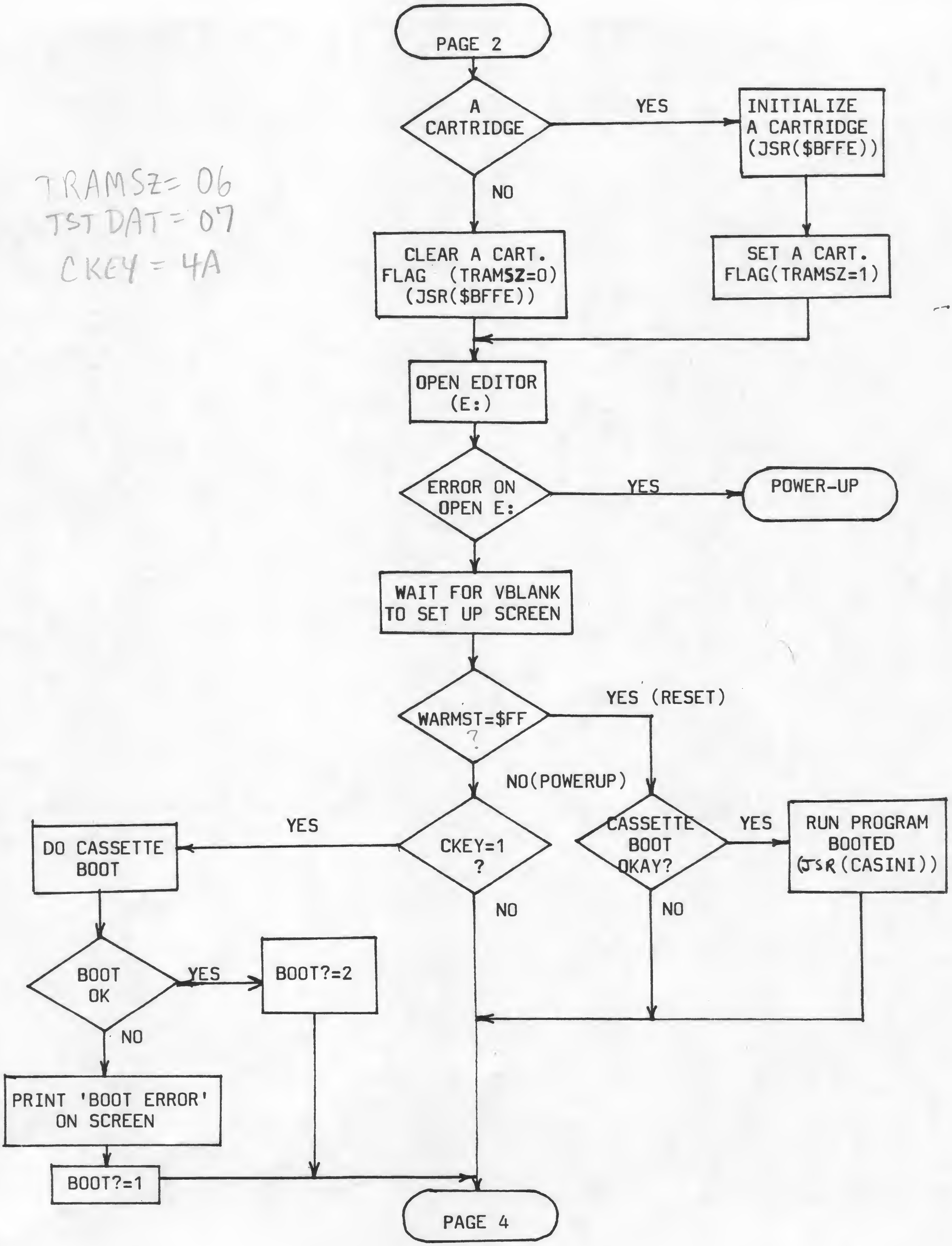
```
1000 REM SPECIFY WHICH TO DELETE
1010 PRINT
1020 PRINT "(0=END) AREA CODE ";:INPUT ACODE$:IF ACODE$="0" THEN 1200
1100 REM SEARCH ACTIVE RECORDS FOR MATCHING AREA CODE
1110 FOR RECORD=1 TO FILEMAX
1115 GOSUB 9600
1120 IF STS=1 THEN GOSUB 5100
1125 NEXT RECORD
1130 GOSUB 9500
1140 GOTO 1000
1200 REM PRINT FILE TO SCREEN
1210 PRINT :PRINT "CODE","LOCATION":PRINT
1220 FOR RECORD=1 TO FILEMAX
1230 GOSUB 9600
1240 IF STS=1 THEN GOSUB 5200
1250 NEXT RECORD
1300 REM ALLOW HARDCOPY
1310 PRINT :PRINT "(Y/N) DO YOU WANT A PRINTED LIST ";:INPUT YN$
1320 IF YN$<>"Y" THEN 4900
1330 LPRINT "CODE","LOCATION":LPRINT
1340 FOR RECORD=1 TO FILEMAX
1350 GOSUB 9600
1360 IF STS=1 THEN GOSUB 5300
1370 NEXT RECORD
1380 GOTO 4900
4900 REM END
4910 CLOSE #CHANNEL
4920 PRINT "END OF EXECUTION"
4930 END
5000 REM PROCESS ACTIVE RECORD/DISPLAY
5010 INPUT #CHANNEL,FILEREC$
5020 IF FILEREC$(1,3)<>ACODE$ THEN RETURN
5030 MATCH=1
5040 PRINT "LOCATION: ";FILEREC$(4)
5050 RETURN
5100 REM PROCESS ACTIVE RECORD/DELETE
5110 INPUT #CHANNEL,FILEREC$
5120 IF FILEREC$(1,3)<>ACODE$ THEN RETURN
5130 GOSUB 9450
5150 PRINT "DELETED ";FILEREC$(4)
5160 RETURN
5200 REM PROCESS ACTIVE RECORD/PRINT
5210 INPUT #CHANNEL,FILEREC$
5220 PRINT FILEREC$(1,3),FILEREC$(4)
5230 RETURN
5300 REM PROCESS ACTIVE RECORD/LPRINT
5310 INPUT #CHANNEL,FILEREC$
5320 LPRINT FILEREC$(1,3),FILEREC$(4)
5330 RETURN
```

```
9300 REM 'FILEOPEN.SUB' WBB 3-30-81
9305 REM OPEN A FILE IN MODE 12 AND DEFINE ALL VARIABLES
9310 OPEN #CHANNEL,12,0,FILE$
9315 DIM FILEHED$(124)
9320 INPUT #CHANNEL,FILEHED$
9325 FILESEC=ASC(FILEHED$(1))*256+ASC(FILEHED$(2))
9330 FILEBYT=ASC(FILEHED$(3))
9335 FILESTS=ASC(FILEHED$(4))
9340 FILEMAX=ASC(FILEHED$(5))*256+ASC(FILEHED$(6))
9345 FILELEN=ASC(FILEHED$(7))*256+ASC(FILEHED$(8))
9350 DIM FILEPTR$(4+4*FILEMAX),FILEREC$(FILELEN)
9355 INPUT #CHANNEL,FILEPTR$
9360 RETURN
9400 REM 'FILEADD.SUB' WBB 3-30-81
9405 REM ALLOCATE THE NEXT AVAILABLE RECORD
9410 RECORD=0
9415 IF FILEMAX=0 THEN RETURN
9420 FOR RECORD1=1 TO FILEMAX
9425 B=RECORD1*4+4
9430 IF FILEPTR$(B,B)=CHR$(0) THEN
RECORD=RECORD1:RECORD1=FILEMAX:FILEPTR$(B,B)=CHR$(1)
9435 NEXT RECORD1
9440 RETURN
9450 REM 'FILEDEL.SUB' WBB 3-30-81
9455 REM DELETE AN ACTIVE RECORD
9460 B=RECORD*4+4
9465 FILEPTR$(B,B)=CHR$(0)
9470 RETURN
9500 REM 'FILEPTR.SUB' WBB 3-19-81
9510 REM WRITE FILEPTR$
9515 S=ASC(FILEPTR$(1))*256+ASC(FILEPTR$(2))
9520 B=ASC(FILEPTR$(3))
9525 POINT #1,S,B
9530 PRINT #1;FILEPTR$
9535 RETURN
9600 REM 'FILEPOS.SUB' WBB 3-31-81
9605 REM POINT FILE TO RECORD
9610 S=ASC(FILEPTR$(RECORD*4+1))*256+ASC(FILEPTR$(RECORD*4+2))
9615 B=ASC(FILEPTR$(RECORD*4+3))
9620 STS=ASC(FILEPTR$(RECORD*4+4))
9625 POINT #CHANNEL,S,B
9630 RETURN
```

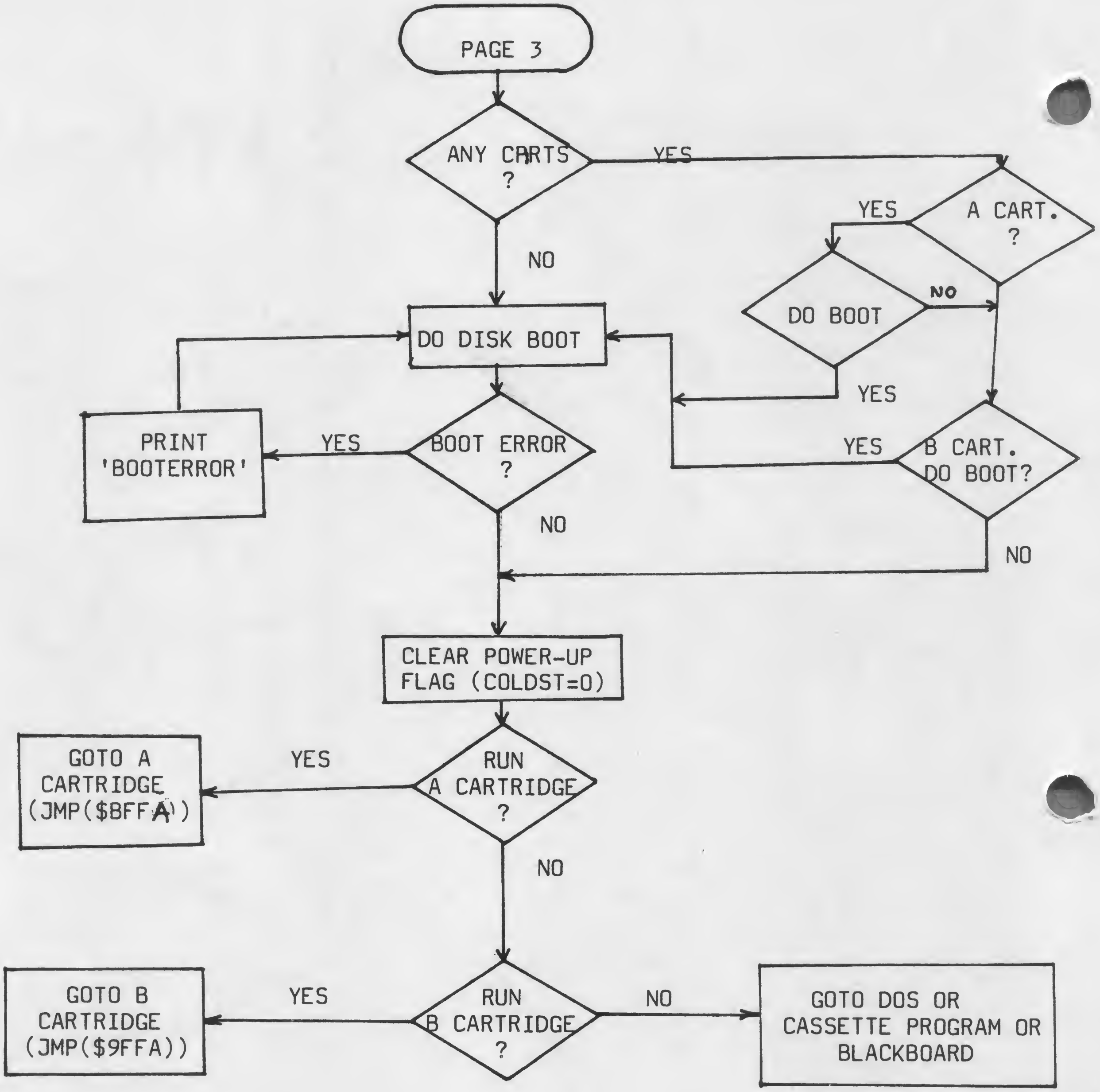
APPENDIX IX - INITIALIZATION


RESET
POWER-UP
SEI
COLDST=0
NO (MIDDLE OF COLDSTART)
YES
WARMST=$FF
SEI
WARMST=0
0008
0009
DIAGNOSTIC CARTRIDGE
YES
GOTO STANDALONE DIAGNOSTIC CARTRIDGE (JMP ($BFFE))
NO
FIND # OF 4K BLOCKS OF RAM (TRAMSZ) INITIALIZE POKEY, ANTIC, CTIA/GTIA
CLEAR O.S. RAM ($200-3FF,$10-7F)
(RESET) NO
WARMST=0
YES(POWERUP)
CLEAR ALL RAM ($08-TRAMSZ) SET DEFAULT CARTRIDGE TO BLACKBOARD (DOSVEC=BLKBDV) SET COLDST TO MIDDLE OF POWER-UP (COLDST=$FF)
SET SCREEN MARGINS (LMARGN=2 RMARGN=39)
MOVE IRQ VECTOR TABLE FROM ROM TO RAM (VDSLST-VVBLKD)
PAGE 2


PAGE 1
CLEAR BREAK KEY (BRKKEY=$FF)
SET MEMORY SIZE
RAMSIZ=TRAMSZ
MEMTOP=TRAMSZ
MEMLO=$700
INITIALIZE DEVICE HANDLERS
EDITOR (E:)
SCREEN (S:)
KEYBOARD (K:)
PRINTER (P:)
CASSETTE (C:)
START KEY
YES
SET CASSETTE BOOT
(CKEY=1)
NO
NO CASSETTE BOOT
(CKEY=0)
ENABLE IRQ INTERRUPTS
MOVE DEVICE HANDLER
TABLE FROM ROM TO RAM
(TBLENT→HATABS)
B
CARTRIDGE
YES
INITIALIZE
B CARTRIDGE
(JSR(9FFE))
NO
CLEAR B CART.
FLAG(TSDAT=0)
SET B CARTRIDGE
FLAG(TSDAT=1)
PAGE 3


PAGE 2
A CARTRIDGE
YES
INITIALIZE A CARTRIDGE (JSR($BFFE))
NO
CLEAR A CART. FLAG (TRAMSZ=0) (JSR($BFFE))
SET A CART. FLAG(TRAMSZ=1)
OPEN EDITOR (E:)
ERROR ON OPEN E:
YES
POWER-UP
WAIT FOR VBLANK TO SET UP SCREEN
WARMST=$FF ?
YES (RESET)
NO(POWERUP)
YES
DO CASSETTE BOOT
CKEY=1 ?
CASSETTE BOOT OKAY?
YES
RUN PROGRAM BOOTED (JSR(CASINI))
NO
NO
BOOT OK
YES
BOOT?=2
NO
PRINT 'BOOT ERROR' ON SCREEN
BOOT?=1
PAGE 4

TRAMSZ= 06
TST DAT = 07
CKEY = 4A


PAGE 3
ANY CARTS ?
YES
NO
A CART. ?
YES
DO BOOT
NO
YES
DO DISK BOOT
BOOT ERROR ?
YES
PRINT 'BOOTERROR'
NO
B CART. DO BOOT?
YES
NO
CLEAR POWER-UP FLAG (COLDST=0)
RUN A CARTRIDGE ?
YES
GOTO A CARTRIDGE (JMP($BFFA))
NO
RUN B CARTRIDGE ?
YES
GOTO B CARTRIDGE (JMP($9FFA))
NO
GOTO DOS OR CASSETTE PROGRAM OR BLACKBOARD

APPENDIX X
GTIA

The GTIA is a new display chip that will someday replace CTIA. Actually it is nothing more than a CTIA with a few more features. It simply provides three additional modes of interpretation of information coming from the ANTIC chip. ANTIC does not require a new mode to talk to GTIA; instead, it uses the high resolution mode $F. GTIA is completely upward compatable with the CTIA. A brief summary of CTIA's features follows so that the differences between CTIA and GTIA can be presented.

The CTIA is designed to display data on the television screen. It displays the playfield, players and missiles, and detects any overlaps or collisions between objects on the screen. CTIA will interpret the data supplied by ANTIC according to six text modes and eight graphics modes. In a static display, it will use the data from ANTIC to display hue and luminance as defined in one of four color registers. The GTIA expands this to use all nine color registers or 16 hues with one luminance or 16 luminances of one hue in a static display.

The three graphics modes of GTIA are simply three new interpretations of ANTIC mode $F, a hi-resolution mode. All three modes affect the playfield only. Players and missiles can still be added to introduce new hues or luminances or to use the same colors and luminances in more than one way. All displays of hues and luminances can still be changed on-the-fly with display list interrupts. The GTIA uses four bits of data from ANTIC for each pixel, called the pixel data. Each pixel is two color clocks wide and one scan line high. Thus, the pixels are roughly four times wider than their height. The display has a resolution of 80 pixels across by 192 down. Each line then requires 320 bits or 40 bytes of memory, the same number of bytes used in ANTIC mode $F. Therefore for a program to run the GTIA modes it must have at least 8K of free RAM for the display.

The GTIA modes are selected by the priority register, PRIOR. PRIOR is shadowed at location $26F hex by the OS and is located at D01B hex in the chip. Bits D6 and D7 are the controlling bits. When neither is set there are no GTIA modes and GTIA operates just like CTIA. When D7 is 0 and D6 is 1, Mode 9 is specified which allows 16 different luminances of the same hue. Remember the pixel data supplied by ANTIC is four bits wide which means 16 different values can be represented. Players and missiles can be used in this mode to introduce additional hues. When D7 is 1 and D6 is 0, Mode 10 is specified. This mode gives nine colors in the display by using the four playfield color registers plus the four player/missile color registers plus the one background color register. When players are used in this mode, the four player/missile color registers are used for them also. When D7 is 1 and D6 is 1, Mode 11 is specified. This mode gives 16 hues with the same luminance again because 16 different values can be represented by four bits. Players and missiles can be used in this mode to introduce different luminances.

PRIOR

D7	D6						

D7	D6	OPTION	
0	0	No GTIA modes (CTIA operation)	(Modes 0-8)
0	1	1 Hue, 16 Luminances	(Mode 9)
1	0	9 Hues/Luminances	(Mode 10)
1	1	16 Hues, 1 Luminances	(Mode 11)

Figure X.1
Bit Pattern in PRIOR selects GTIA

Setting up the new GTIA modes is as simple as setting up the present modes used in CTIA. To implement the modes from BASIC simply use the commands GRAPHICS 9, GRAPHICS 10, and GRAPHICS 11 for Mode 9, Mode 10, and Mode 11 respectively. In Assembly Language selecting one of these modes is identical to opening the screen for any of the other modes. If you are building your own display list then PRIOR must be set to select the correct mode as in Figure X.1.

Mode 9 produces up to 16 different luminances of the same hue. ANTIC provides the pixel data which selects one of 16 different luminances. The background color register provides the hue. In BASIC this is done using the SETCOLOR command to set the hue value in the upper nybble of the background color register, and to set the luminance value in the lower nybble to all zeroes. The format of the command is

```
SETCOLOR 4,hue value,0
```

where 4 specifies the background color register, "hue value" sets the hue and can be anything from 0 to 15, and 0 will set the luminance part of the register to zero. This has to be done because the pixel data from ANTIC will then be logically OR'ed with the lower nybble of the background color register to set the luminance that appears on the screen. The COLOR command is then used to select luminances for drawing on the screen by using values from 0 to 15 as its parameter. So a BASIC program will include at least the following statements to use Mode 9:

```
GRAPHICS 9          to specify Mode 9
SETCOLOR 4,12,0     to initialize the background color
                    register to some hue, in this case green.
FOR I= 0 TO 15      some method where the
COLOR I             COLOR command is used
```

```
        PLOT 4,I+10        to vary luminance.
        NEXT I
```

In Assembly Language use the OS shadow for the background color register $2C8 to set the hue in the upper four bits with hex values from $0 to $F. If CIO calls are used, store the pixel data into the OS register ATACHR located at $2FB. This selects the luminance with hex values from $0 to $F. If you are maintaining your own display data then the pixel data goes directly into the left or right half of the display RAM byte.

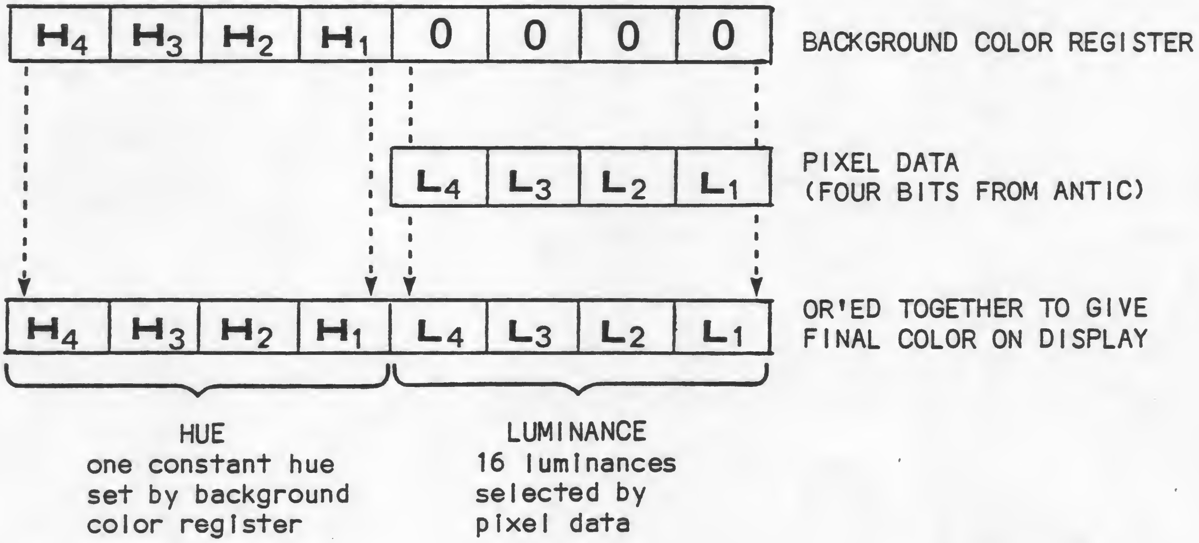


Figure X.2
Background Color Register OR'ed with pixel data to give final color.

Mode 11 is similar to Mode 9 except that it provides 16 different hues all with the same luminance. Again ANTIC will provide the pixel data to select one of 16 different hues. In BASIC the SETCOLOR command is used to set up the single luminance value in the lower nybble of the background color register, and in the upper nybble, the hue value will be set to all zeroes. The format of the command is

```
SETCOLOR 4,0,luminance value
```

where 4 specifies the background color register, 0 sets the upper nybble to zero and "luminance value" sets the value of the luminance and can range from 0 to 15. As with the other graphics modes (except Mode 9), the first bit of the luminance is not used, so effectively only even numbers result in distinct luminances which gives eight different possible luminances in this mode. The COLOR command is used in this mode to select the various hues by using values from 0 to 15 in its parameter. The pixel data from ANTIC will be logically OR'ed with the upper nybble of the background color register to set the hue part of the value that ultimately generates the color on the screen. So a BASIC program using Mode 11 will include at least the following statements:

```
        GRAPHICS 11          to specify Mode 11
        SETCOLOR 4,0,12      to initialize the background
                             color register to some luminance,
                             in this case very bright
        FOR I= 0 TO 15       some method where the
        COLOR I              COLOR command is used
        PLOT 4,I+10          to vary the hue
        NEXT I
```

In Assembly Language use the OS shadow for the background color register $2C8 to set the luminance in the lower four bits with hex values from $0 to $F. If CIO calls are used, store the pixel data into ATACHR located at $2FB. This selects the hue with hex values from $0 to $F. If you are maintaining your own display data then the pixel data goes directly into the left or right half of the display RAM byte.

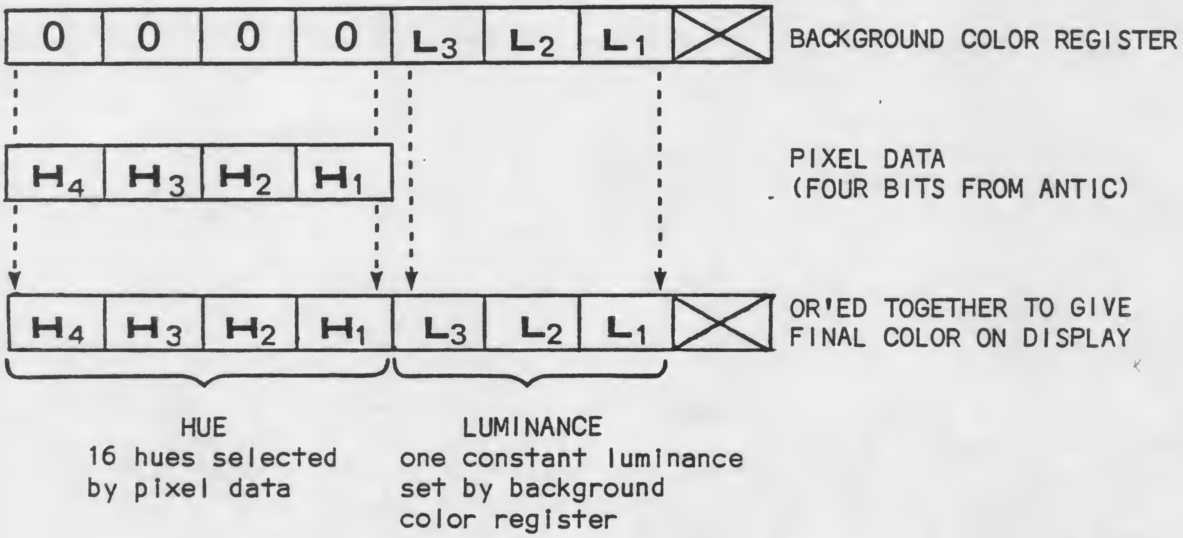

0 0 0 0 L_3 L_2 L_1
BACKGROUND COLOR REGISTER
H_4 H_3 H_2 H_1
PIXEL DATA (FOUR BITS FROM ANTIC)
H_4 H_3 H_2 H_1 L_3 L_2 L_1
OR'ED TOGETHER TO GIVE FINAL COLOR ON DISPLAY
HUE
16 hues selected by pixel data
LUMINANCE
one constant luminance set by background color register

Figure X.3
Background Color Register OR'ed with pixel data to give final color.

Mode 10 will allow all nine color registers to be used in the playfield at one time. Each color register to be used must be set to some combination of hue and luminance. The pixel data from ANTIC is used in this mode to select one of the color registers for display. In BASIC the SETCOLOR command can be used as described in the BASIC Reference Manual to set the colors in the background and the four playfield registers. These can also be set by using the POKE instruction to addresses 708-712 where the four playfield registers and the background register are located. The POKE instruction must be used to set the four player/missile color registers at locations 704-707. The COLOR command is used to select the color register desired. The only meaningful values for its arguement are 0 to 8. A problem arises with this mode. ANTIC supplies four bits of data per pixel, as it does with Modes 9 and 11. This allows for the selection of 16 color registers. However, only nine color registers exist in the hardware. An illegal data value between 9 and 15 will select one of the lower value color registers. A BASIC program using mode 10 will include:

1) a GRAPHICS 10 command to specify Mode 10;
2) a set of POKE instructions to put hues and luminances into the color registers,
OR a combination of SETCOLOR commands and POKE instructions to do that;
3) a COLOR command to select the desired color register.

In Assembly Language, store the pixel data in ATACHR ($2FB) or directly into the display RAM byte as in Modes 9 and 11. In this mode the pixel data can range from 0 to 8 and selects one of the nine color registers.

COLOR STATEMENT VALUE	COLOR REGISTER USED	OS SHADOW
0	D012	2C0
1	D013	2C1
2	D014	2C2
3	D015	2C3
4	D016	2C4
5	D017	2C5
6	D018	2C6
7	D019	2C7
8	D01A	2C8

Figure X.4
Color Register numbers and locations and COLOR command reference.

An important question arises in conjuction with GTIA concerning compatability. GTIA is fully upward compatible with the CTIA and all software that runs on a CTIA system will run the same way on a system with GTIA. This means you still have the full use of players and missiles, still have collision and overlap detection and display list interrupts. The GTIA graphics modes are fully supported by the OS and all graphics commands and utilities that run in the CTIA modes can be used in GTIA modes.

More colors are available to display at one time on the screen. Sixteen color changes can occur on one line totally independent of processor intervention. This is actually better than what could be done with display list interrupts which could give at most only 12 color changes per line. Much finer contour and depth can be represented using the shading available in Mode 9. This means three dimensional graphics can be realistically displayed.

On the other hand, there are some disadvantages. GTIA modes are map modes, there can be no text displayed in these modes. A custom display list must be used to switch to a mode that supports character displays. The GTIA pixel is a long, skinny horizontal rectangle (4:1, width to height) and does not represent curved lines well. Because each pixel uses four bits of information, GTIA requires nearly 8K of free RAM to operate. Although it is upward compatable, it is NOT downward compatable. Thus programs which use GTIA modes will NOT produce correct displays on computers that have CTIA's. They may well be recognizable but will not be as colorful. There is no way currently for a program to determine whether or not a GTIA is present in a system.

GLOSSARY

$

The symbol which indicates a number should be interpreted as hexadecimal.

ANTIC

This is a separate microprocessor contained within the ATARI 400/800 which is dedicated to the television display. ANTIC is user-programmable with an instruction set, a program (the "display list"), and data (the "display memory").

ATTRACT MODE

This is a feature provided by the operating system which, after nine minutes without a key being pressed, cycles the colors on the screen through random hues at lowered luminances. This insures that a computer left unattended for several hours doesn't burn a static image into the television screen.

BACKGROUND

The area of the television screen display upon which player-missile graphics objects or playfield objects and/or text are projected. Background has its own user-definable color.

BORDER

In BASIC Mode 0, this is the area of the television screen display which is formed by the four edges of the screen. The border takes background color.

BRKKEY

A flag set when the OS senses that the BREAK key is typed. BRKKEY's normal value is $FF -- if it changes, then the BREAK key has been typed.

BYTE COUNT

This is the file pointer's position within a sector on diskette.

CASSETTE BOOT FILE

A standard or user-created file which boots from cassette at power-up or SYSTEM RESET.

CHARACTER GRAPHICS

The technique of redefining the individual characters of a character set to form graphics images instead of text characters.

CHARACTER IMAGE

The unique 8 X 8 pixel grid which defines a particular character's shape.

CHARACTER MODE

This is a specific type of ANTIC display mode which displays screen display memory data bytes as characters, using a character set. There are 6 ANTIC character modes, 3 of which are accessible from BASIC.

CHARACTER NAME BYTE

A one-byte ANTIC display memory value which selects a unique character within the current character set by the character's numerical position in that set.

CHARACTER SET INDIRECTION

The technique of specifying to ANTIC a particular character set to be used by placing that set's beginning page address into CHBAS.

CHBAS

The OS shadow location which ANTIC uses to find the current character set which is to be used for character display modes. CHBAS is at decimal address 756.

CHECKSUM

This is a single byte sum of all the bytes in a record (either disk I/O or cassette I/O). For cassette I/O this includes addition of the two marker characters, computed with end-around carry.

CIO

Acronym for Central I/O system routine. CIO routes I/O control data to the correct device handler and then passes control to the handler. CIO is also the common entry point for most of the OS I/O functions.

COARSE SCROLLING

The process of altering the display list LMS (Load Memory Scan) address bytes in order to vertically or horizontally scroll the screen image, one byte at a time. This is accomplished by adding 1 to or subtracting 1 from the LMS address bytes.

COLDSTART

Synonym for the power-up process which performs a series of system database initializations when the computer power switch is turned on. After coldstart, the system surrenders control to the user.

COLLISION

This occurs when a player image in player-missile graphics coincides with another image. There are 60 possible collisions and each one has a bit assigned to it that can be checked. These bits are mapped into 16 registers in CTIA (with only the lower 4 bits used).

COLLISION DETECTION

Primarily of value for games. This is a hardware-implemented ability to detect if any one of the 60 player-missile collision possibilities has occurred.

COLOR

One of 128 values obtained from a hue-luminosity combination which is stored in a color register.

COLOR CLOCK

The standard unit of horizontal distance on the television screen. There are 228 color clocks in a horizontal scan line.

COLOR REGISTER

A hardware register (with corresponding OS shadow location) used to add color to various portions of the screen display. There are 9 color registers available on the ATARI Personal Computer System.

COLOR REGISTER INDIRECTION

The technique of specifying a particular color by coding its value into a color register.

COLOR SIGNAL

This contains the color information which is combined with the primary signal to form the modulating television signal. The Color Signal oscillates at 3.579 MHZ.

COLRSH

A zero-page location ($4F) set up and updated by the OS during vertical blank interrupts for ATTRACT mode processing. When ATTRACT mode is in force, COLRSH is given a new random value every four seconds.

COMMAND

In BASIC, this is the first executable token of a BASIC statement that tells BASIC to interpret the tokens that follow in a particular way.

CONSTANT

In BASIC, this is a six-byte BCD value preceeded by a special token. This value remains unchanged throughout the program execution.

CONTROL BYTE

In cassette I/O, this is part of every record. It contains one of three possible values.

CTIA

A television interface chip which is controlled primarily by ANTIC. CTIA converts ANTIC's digital commands into a signal that is sent to the television.

CURRENT STATEMENT

In BASIC, this is the current token within a line of the Statement Table.

CYCLE STEALING

This occurs when ANTIC interrupts 6502 processing in order to perform DMA functions for screen display purposes.

CYCLIC ANIMATION

The technique of repetitively flipping through colors, graphics images, or character graphics sets to animate screen images.

DCB

Acronym for Device Control Block. The DCB is used by the I/O subsystem to communicate between the device handlers and SIO.

DEVICE HANDLERS

Routines present in OS ROM which are called through CIO (as long as the handler has an entry in HATABS) to communicate to particular devices. Currently supported are the Display Editor, the Screen, the Keyboard, the Printer, and the Cassette.

DEVICE SPEC

A special HATABS code which specifies a particular I/O device.

DIAGONAL SCROLLING

This results from the combination of horizontal and vertical scrolling of the screen image.

DISPLAY FORMAT

A screen image on paper which is translated into a sequence of mode lines which themselves eventually get translated into ANTIC's display list instructions.

DISPLAY LIST

ANTIC's "program" defined by the user or provided automatically (through a GRAPHICS command) by BASIC. The display list specifies where the screen data may be found, what display modes to use to interpret screen data, and what special display options (if any) should be implemented.

DISPLAY LIST INTERRUPT

A special ANTIC display list instruction which interrupts the 6502 microprocessor during the drawing of the screen image, allowing the 6502 to change the screen parameters.

DISPLAY MODE

Either a BASIC or ANTIC methodology for interpreting text or map data bytes in screen memory and displaying them on the screen.

DLI VECTOR

This is a two-byte pointer (lo byte, high byte) to the Display List Interrupt service routine. This vector is set by the user and is located at [512,513] decimal.

DMA

Direct Memory Access. This occurs when ANTIC or the 6502 fetch an instruction byte or data byte from memory.

DMACTL

The hardware register whose bit settings control, among other things, player vertical resolution and player-missile graphics enabling.

DOS

Acronym for Disk Operating System which is an extension of the OS that allows the user to access disk drive mass storage as files.

DOUBLE-LINE RESOLUTION

A unit of vertical resolution for a player in player-missile graphics. Each player byte occupies two horizontal scan lines on the screen, and each player table is 128 bytes long.

DRKMSK

A zero-page ($4E) location set up and updated by the OS during vertical blank interrupts for ATTRACT mode processing. Usage of DRKMSK lops off the highest luminance bit of the right nybble of a color register's value. This insures a low luminance for ATTRACT mode.

DUP

Acronym for Disk Utility Package. DUP is a set of utilities for diskette, familiarly seen as the DOS menu. DUP executes commands by calling FMS through CIO.

DYNAMIC DISPLAY LIST

This is an ANTIC display list which the 6502 changes during vertical blank periods, allowing for even a greater degree of flexibility in the screen display.

EOL

In BASIC, "End-of-Line", a character with the value $9B.

FILE

In cassette I/O, this consists of a 20-second leader of the mark tone plus any number of data bytes, and end-of-file. In diskette I/O this consists of a number of sectors linked by pointers (125 data bytes per sector).

FILE POINTER

For diskette I/O, this is a value which indicates the current position in a file by specifying the Sector Number and the Byte Count. DOS keeps a file pointer for every file currently open.

FINE SCROLLING

The process of horizontally or vertically scrolling a screen image in color clock or scan line increments. The horizontal scrolling and vertical scrolling hardware registers must be used to fine scroll.

FMS

File Manager System. FMS is a non-resident device handler which supports some special CIO functions.

FONT

A collection of characters which constitutes a character set. These characters can be either text or graphics images.

FOREGROUND

Equivalent to playfield, the area of the screen which directly overlays the background of the screen. Foreground is formed by map displays and/or text.

FORMAT

A resident disk handler command that clears all the tracks on diskette.

FUNCTION

In BASIC, a token that when executed returns a value to the program.

GRAPHICS INDIRECTION

A special feature of the ATARI system which allows color register and character set generality by using indirect pointers to color and character set values.

HATABS

The device handler entry point table which is used by CIO. HATABS is located at $031A.

HORIZONTAL BLANK

This is the period during which the electron beam (as it draws the screen image) returns from the right edge of the screen to the left edge.

HORIZONTAL POSITION REGISTER

A specialized register which contains a user-definable value for the horizontal position of a player in player-missile graphics. This value is in units of color clocks.

HORIZONTAL SCAN LINE

The fundamental unit of measurement of vertical distance on the screen. The scan line is formed by a single trace of the electron beam across the screen.

HORIZONTAL SCROLL ENABLE BIT

This is bit D6 of the ANTIC display instruction which enables horizontal scrolling through the HSCROL register.

HORIZONTAL SCROLLING

This is the process of "sliding" the screen "window" to the left or right over display memory in order to display more information than could be seen with a static screen. Either coarse or fine horizontal scrolling is available.

HSCROL

This is the horizontal fine scrolling register located at $D404, containing the number of clocks by which a line is to be horizontally scrolled.

HUE

The upper nybble value of a color register's color. There are 16 possible hues ($0 to $F) which in combination with a luminosity value constitute distinct "colors." Examples of hues are "black", "red", and "gold."

I/O SUBSYSTEM

A set of system routines that interface to the I/O hardware.

I/O SYSTEM CONTROL BLOCKS

These blocks are elements of the I/O subsystem which are used to communicate information about the I/O function to be executed.

IMMEDIATE MODE

In BASIC, the mode where the input line is not preceeded by a line number. BASIC immediately executes the line.

INPUT BAUD RATE

For cassette I/O, this is assumed to be a nominal 600 baud (physical bits per second). However, this rate is adjusted by SIO to account for drive motor variations, stretched tape, etc.

INPUT LINE BUFFER

In BASIC, from $580 to $5FF.

INTER-RECORD GAP

For cassette I/O records, this consists of the Post-record gap of a given record followed by the Pre-record Write Tone of the next record.

INTERIM MASTER

In the mass production of cassette tapes, this is the backup copy for the Work Master tape.

IOCB

Acronym for Input/Output Control Block. There are eight of these whose function is to communicate between the user programs and CIO.

IRQ

Maskable (can be enabled or disabled by the 6502) interrupts such as the Break Key IRQ.

IRQEN

The write-only register that contains the IRQ enable/disable bits. IRQEN is shadowed at POKMSK.

KERNEL

A primitive software/hardware technique which consists of a 6502 program loop which is precisely timed to the display cycle of the television set. The kernel code monitors the VCOUNT register and consults a table of screen changes catalogued as a function of VCOUNT values so that the 6502 can arbitrarily control all graphics values for the entire screen.

LINE

In BASIC, a line consists of one or more BASIC statements preceeded either by a line number in the range of 0 to 32767, or an immediate mode line with no line number.

LOMEM

In BASIC, this is the pointer ([80,81] decimal) to a buffer used to tokenize one line of code. The buffer is 256 bytes long, residing at the end of the operating system's allocated RAM.

LUMINANCE

The lower nybble of a color register's color. There are 8 even-numbered values for luminance ($0 to $F, even values only) which in combination with hue values produce the 128 "colors" available on the ATARI 400/800.

MAP MODE

This is a specific type of ANTIC display mode using simple colored screen pixels instead of characters for the screen display. There are 8 ANTIC map modes, with varying degrees of resolution. Six of these are callable from BASIC.

MARK

For cassette I/O, this is a 5327 HZ frequency.

MARKER CHARACTER

For cassette I/O, this is a 55 (hex) value whose purpose is for adjusting the baud rate. Including the start and stop bits, each marker character is 10 bits long.

MASTER TAPE

In cassette tape mass production, this is an open-reel, 1/4 track, 1/4 inch tape recorded at 7 1/2 inches per second. The Master Tape becomes the Source Master prior to the duplication process.

MEMTOP

In BASIC, a pointer ([90,91] decimal) to the top of application RAM, the end of the user program. Program expansion can occur from this point to the end of free RAM, which is defined by the start of the display list. This MEMTOP is not the same as the OS variable called MEMTOP.

MISSILE

A one-dimensional image in RAM used in player-missile graphics which is 2 bits wide. There is a maximum of 4 missiles, one for each player.

MISSILE COLOR

The color of a missile in player-missile graphics. Each of the four missiles takes on the color of its associated player.

MODE LINE

A collection of horizontal scan lines for screen displays. Depending upon the BASIC or ANTIC display mode in effect, a mode line will be composed of varying numbers of scan lines. By the same token, depending upon the display mode, a screen image will be composed of varying numbers of mode lines.

MONITOR

A program in ROM that handles both the system power-up and SYSTEM RESET sequences.

N-BIT

A 6502 processor status register bit which is set by, among other things, I/O calls to indicate the sucess or failure of an I/O operation.

NARROW PLAYFIELD

A screen display width option equal to a width of 128 color clocks.

NMI

Non-Maskable Interrupt (i.e., cannot be disabled by the 6502). The Display List Interrupt and the Vertical Blank Interrupt are both NMIs. These can be disabled with the ANTIC NMIEN register.

NMIEN

The Non-Maskable Interrupt Enable reister which controls enabling of various interrupts such as the Display List Interrupt (DLI).

NORMAL IRG MODE

In cassette I/O, this is a mode where the tape always comes to a stop after each record is read. If the computer stops the tape and gets its processing done fast enough, then the next read may occur so quickly that the cassette deck may see only a slight dip in the control line.

NORMAL PLAYFIELD

A screen display width option equal to a width of 160 color clocks.

OPERATOR

In BASIC, any one of the 46 tokens that in some way move or modify the values that follow them.

OPERATOR STACK

In BASIC, a software stack where operators are placed when an arithmetic BASIC expression is being evaluated.

OVERSCAN

The "spreading out" of a television image by the raster scan method of display so that the edges of the picture are off the edge of the television tube. This guarantees no unsightly borders in the television picture.

PIXEL

The standard point-unit of vertical distance on the television screen. The normal limit of a television set used with the ATARI 400/800 is 192 pixels vertically.

PLAYER

A one-dimensional RAM image used in player-missile graphics which can be 128 bytes (double-line resolution) or 256 bytes (single-line resolution) long. The player appears as a vertical band 8 pixels wide stretching from the top of the screen to the bottom. There is a maximum of four independent players.

PLAYER COLOR

The color of a player in player-missile graphics. Each of the four independent players has its own color stored in its associated color register.

PLAYER-MISSILE AREA

A RAM area which contains the images of the four players and four missiles of player-missile graphics, as well as some extra RAM. The player-missile area must be on a 1K boundary for single-line resolution players or a 2K boundary for double-line resolution players.

PLAYER-MISSILE GRAPHICS

Atari's solution for simplifying animation by creating an image (a 'player' or 'missile') which is one-dimensional in RAM but two-dimensional on the screen.

PLAYFIELD

The area of the screen which directly overlays the background of the screen. Map graphics and/or text form this playfield.

PLAYFIELD ANIMATION

The technique of animating an object by moving its two-dimensional image bytes to new locations in screen memory, and then erasing the defining bytes of the old image before displaying the newly-moved image.

PMBAS

A register which points to the beginning of the player-missile area.

POKEY

A digital I/O chip which handles the serial I/O bus, audio generation, keyboard scan, and random number generation. POKEY also digitizes the resistive paddle inputs and controls maskable interrupt (IRQ) requests.

POKEY TIMERS

Unlike System Timers, the POKEY chip timers are clocked by frequencies set by the user.

POST-RECORD GAP

A pure mark tone frequency used as a post-record delimiter in cassette I/O.

PRE-RECORD WRITE TONE

A pure mark tone frequency used as a pre-record delimiter in cassette I/O.

PRIMARY SIGNAL

This contains the luminance information -- brightness data, horizontal and vertical syncs and blanks -- of the modulated television signal.

PRIORITY-CONTROL REGISTER

Also known as PRIOR, and shadowed at GPRIOR. This register specifies which playfield, player, or background images have priority in the case of image overlaps during the screen display process.

RAM VECTOR

Alterable system vector that contains two-byte addresses to system routines, handler entry pointers, or to initialization routines. RAM vectors are initialized at power-up and SYSTEM RESET.

RASTER SCAN

A television display system which uses an electron beam generated at the rear of the television tube. The beam sweeps across the screen in a regular left-to-right, top-to-bottom fashion.

RECORD

For diskette I/O, a group of bytes delimited by EOLs ($9B). For cassette I/O, this is a group of 132 bytes which is composed of 2 marker characters for cassette speed measurement, a control byte, 128 data bytes, and the checksum byte.

REDEFINED CHARACTER SET

Any user-defined character set designated for use by ANTIC other than the standard character set in ROM. CHBAS (decimal address 756) contains the beginning page address of any redefined character set.

RESIDENT DISK HANDLER

This software performs five important low-level disk I/O functions such as FORMAT, READ SECTOR, WRITE SECTOR, WRITE/VERIFY SECTOR, and STATUS.

ROM VECTOR

Unalterable system vector that contains JMP instructions to system routines.

RTCLOK

One of the system timers which is 3 bytes in length and is updated during immediate VBLANK. RTCLOK can be used as an application's timepiece.

RUNSTK

In BASIC, a pointer ([8E,8F] decimal) to the Run Time Stack.

RUN TIME STACK

In BASIC, a software stack which contains GOSUB and FOR/NEXT return address entries.

SCREEN MEMORY

A RAM area used by the 6502 to store bytes of data that will be fetched (by DMA) by ANTIC to be interpreted and eventually displayed as images on the screen.

SECTOR

On a diskette, this is a 128-byte physical area. The diskette contains 40 tracks with 18 sectors per track.

SECTOR NUMBER

A value from 1 to 719 which designates to which diskette sector the file pointer is currently pointing.

SETVBV

A system routine which, among other functions, correctly sets the system timers and sets user-definable interrupt vector addresses.

SHADOWING

The process whereby a software location is used by the OS to update hardware registers during vertical blank periods.

SHORT IRG MODE

In cassette I/O, this means the tape is not stopped between records. The BASIC commands "CSAVE" and "CLOAD" both specify this mode.

SINGLE-LINE RESOLUTION

A unit of vertical resolution for a player in player-missile graphics. Each player byte occupies one horizontal scan line on the screen, and each player table is 256 bytes long.

SIO

Serial I/O system routine which handles communication between the serial device handlers in the computer and the serial bus.

SIO INTERRUPTS

These are 3 IRQ interrupts used by SIO to send and receive serial bus communications to serial bus devices. These 3 are VSERIR (Serial Input Ready), VSEROR (Serial Output Needed), and VSEROC (Transmission Finished).

SOUND REGISTER

Audio-producing hardware in the ATARI Personal Computer System which contains frequency, volume, and distortion information, but not duration.

SOURCE MASTER

In the mass production of cassette tapes, this is identical to the Master Tape.

SPACE

For cassette I/O, this is a 3995 HZ frequency output to the cassette tape as a delimiter in conjunction with mark tones.

STANDARD CHARACTER SET

The default character set resident in ROM which is used by the ATARI 400/800.

STARP

In BASIC, the pointer ([8C,8D] decimal) to the String Array Area.

STATEMENT

In BASIC, this is a complete "sentence" of tokens that causes BASIC to perform some meaningful task. In LIST form, statements are separated by colons.

STATEMENT TABLE

In BASIC, this is a block of data which includes all the lines of code that have been entered by the user and tokenized by BASIC. This table also includes the immediate mode line.

STMCUR

In BASIC, the pointer ([8A,8B] decimal) to the current BASIC statement.

STMTAB

In BASIC, this is the pointer ([88,89] decimal) to the Statement Table.

STRING ARRAY AREA

In BASIC, this block contains all the string and array data.

SYNC MARK

This is a 3995 HZ Space frequency used as a sort of "end-of-record" marker for audio tracks on the cassette. In applications software it is useful for synchronizing the computer screen display with cassette audio.

SYSTEM DATABASE

This is an area which occupies RAM Pages 0 through 4, containing many locations that store information of interest to the user.

SYSTEM TIMER

A timer provided by the ATARI 400/800 that runs at the frequency of the television frame which for North American T.V.s (NTSC) is 59.923334 HZ. European (PAL) televisions run at 50 HZ. There are 6 system timers, and they are clocked as part of the vertical blank process.

TELEVISION ARTIFACT

This refers to a spot or "pixel" on the screen that displays a different color than the one assigned to it. "Artifacting" is possible in ANTIC modes 2, 3, and 15 which correspond to BASIC modes 0, no mode, and 8.

TEXT WINDOW

On a screen display, this is a two-dimensional area set aside for textual I/O with the user.

TOKEN

In BASIC, an 8-bit byte containing a particular execution code.

TOKENIZING

In BASIC, this is the process of getting a line of input and creating a series of 8-bit bytes which contain tokens, meaningful execution codes.

VARIABLE

In BASIC, a token that is an indirect pointer to its actual value.

VARIABLE NAME TABLE

In BASIC, this is the table containing a list of all the variable names that have been entered in a program.

VARIABLE VALUE TABLE

In BASIC, this table contains current information on each variable.

VBREAK

This is the 6502 BRK instruction IRQ vector. Whenever a $00 opcode (the software break instruction) is executed, this interrupt occurs. VBREAK normally points to an RTI instruction.

VCOUNT REGISTER

The ANTIC register which keeps track of which horizontal scan line ANTIC is displaying.

VDSLST

This is the Display List Interrupt NMI vector located at [$0200,$0201].

VERTICAL BLANK

The period during which the electron beam (as it draws the screen image) returns from the bottom of the screen to the top. This period is about 1400 microseconds in duration.

VERTICAL BLANK INTERRUPT

A non-maskable interrupt which occurs every 60th of a second during the vertical blank time of the television display. During this interrupt, the OS performs various housekeeping functions such as shadowing color registers.

VERTICAL SCROLL ENABLE BIT

This is bit D5 of the ANTIC display list instruction byte which enables vertical fine scrolling through VSCROL ($D405), the vertical fine scroll register.

VERTICAL SCROLLING

The process of vertically "rolling" the display screen "window" over a larger amount of screen data in display memory than can be displayed by a static screen window. Either coarse or fine vertical scrolling is available on the ATARI 400/800.

VIMIRQ

This is the immediate IRQ vector. All IRQs vector through this location. VIMIRQ normally points to the IRQ handler. This vector can be 'stolen' to do user IRQ interrupt processing.

VINTER

This is the Peripheral Interrupt IRQ vector. The interrupt line is also available on the serial bus. VINTER normally points to an RTI instruction.

VKEYBD

This is the keyboard IRQ vector which is activated by pressing any key except BREAK. This vector normally points to the OS's own keyboard IRQ routine.

VNTD

In BASIC, this is the pointer ([84,85] decimal) to the Variable Name Table Dummy end. BASIC uses this pointer to indicate the end of the name table. This pointer normally points to a dummy zero byte when there are less than 128 variables. When 128 variables are present, this points to the last byte of the last variable name.

VNTP

In BASIC, the pointer ([82,83] decimal) to the Variable Name Table.

VPRCED

This is the Peripheral Proceed IRQ vector. The proceed line is available to peripherals on the serial bus. This IRQ is unused at the present and normally points to an RTI instruction.

VSCROL

This is the vertical fine scroll register located at $D405. Into VSCROL the user stuffs the number of scan lines by which the screen line is to be vertically scrolled.

VSERIN

This is the POKEY serial Input Ready IRQ vector.

VSEROR

This is the POKEY serial Output Ready IRQ vector.

VTIMR1

This is the POKEY timer 1 IRQ vector.

VTIMR2

This is the POKEY timer 2 IRQ vector.

VTIMR4

This is the POKEY timer 4 IRQ vector.

VVBLKD

This is the Vertical Blank Deferred NMI interrupt vector located at [$0224,$0225].

VVBLKI

This is the Vertical Blank Immediate NMI interrupt vector located at [$0222,$0223].

VVTP

In BASIC, this is the pointer ([86,87] decimal) to the Variable Value Table.

WARMSTART

Another name for SYSTEM RESET which is similar (in the vector initializing functions) to, but not identical to, a coldstart.

WIDE PLAYFIELD

A screen display width option equal to a width of 192 color clocks.

WORKMASTER

In the mass production of cassette tapes, this is the final master tape from which numerous cassette tapes will actually be manufactured.

WSYNC

Wait for Horizontal Sync of the electron beam which is drawing the screen image. The WSYNC register, when written to in any way, pulls down the RDY line on the 6502 microprocessor, freezing the 6502 until the electron beam drawing the screen image returns to the left edge of the screen.

ZERO-PAGE

In the ATARI Personal Computer System, this is the stretch of memory which spans locations $0000 to $00FF.

ZIOCB

Zero-page I/O Control Block -- used to communicate I/O control data between CIO and the device handlers.